MAGAZINE

BSD

FOR NOVICE AND ADVANCED USERS

BSD Special

Best of David Carlier articles

Cloud service

from a developer point of view

NetBSD and pkgsrc-wip

The Journey of a C Developer

in FreeBSD's World

Kernel and syscalls

Intorduction

Development Tools on FreeBSD

VOL 10 NO 07
ISSUE 07/2016 (83)
1898-9144

EDITORS' WORD

Dear Readers,

Today we are proud to deliver a special BSD Mag issue with compilation of all the best David Carliers articles. David has been cooperating with us for a couple of years. During this time he has delivered a great number of very technical articles, with thousands lines of code. His articles have always been in line with issue themes and we have a feeling that whatever topic we would ask for, he is able to write about it. He possesses great knowledge and passion to open source systems, mostly BSD - of course.

We have chosen his best articles about FreeBSD, NetBSD, NodeJS, HardnedBSD, FreeBSD Security and Cloud.

If you have been wondering who David actually is, at the end of this issue you will find his bio.

We hope you will enjoy the reading.

Marta & BSD Team

MAGAZINE BSD

Editor in Chief:

Marta Ziemianowicz

marta.ziemianowicz@software.com.pl

Contributing:

David Carlier

Top Betatesters & Proofreaders:

Annie Zhang, Denise Ebery, Eric Geissinger, Luca Ferrari, Imad Soltani, Olaoluwa Omokanwaye, Radjis Mahangoe, Mani Kanth and Mark Von-Fange.

Special Thanks:

Annie Zhang

Denise Ebery

DTP:

Marta Ziemianowicz

Senior Consultant/Publisher:

Paweł Marciniak

pawel@software.com.pl

CEO:

Joanna Kretowicz

joanna.kretowicz@software.com.pl

Publisher:

Hakin9 Media SK 02-676 Warsaw, Poland Postepu 17D Poland worldwide publishing editors@bsdmag.org www.bsdmag.org

Hakin9 Media SK is looking for partners from all over the world. If you are interested in cooperation with us, please contact us via e-mail: editors@bsdmag.org.

CONTENTS

CONTENTS

Using the FreeBSD's procstat API in a web context

Among all the numerous specific features of FreeBSD, there is a famous command line to dump the statistics of the various current processes, procstat. Its internal API is fortunately exposed via the well named libprocstat library. Let's imagine we want to display it via a web page so for this article, we re going to use CppCms, one of the good quality C++ web development frameworks with a current FreeBSD 10.2 release version.

1. Procstat API

The list of the available functions can be viewed in this page

https://www.freebsd.org/cgi/man.cgi?query=libprocstat&sektion=3&apropos=0&manpath=FreeBSD%2010.0-RELEASE

We just need to include the necessary headers and to link our application to the shared library libprocstat, simply. For our basic procstat service, we will expose the pids, the paths of the processes and the owners of those.

2. CppCms

We could have used an usual full PHP solution, calling procstat utily via a system call, possibly parsing the output and displaying it. However, doing web development via low level langages is also possible especially in the embedded environments where the resources usage count.

CppCms has a package, so pkg install cppcms (or via the ports) is sufficient. This framework has a lot of useful features, session handling, caching, native encoding handling. For our basic usage, we'll use their advanced template system with the addition of jQuery to make it more appealing.

3. Content

Let's start with the template's content. For this purpose we need a C++ prototype and a CppCms template file.

proclist.h :

```
#include <cppcms/view.h>
#include <vector>


// Just a plain struct to hold a specific process data
struct Procinfo {
        pid_t pid;
        std::string pathName;
        std::string args;
        std::string userName;
        std::string userFullName;
        std::string userHome;
};
// This class will be used by the template's file
// The main CppCms app will fill in the list of processes before the template's rendering
namespace content {
        struct ProcinfoContent : public cppcms::base_content {
                std::vector<Procinfo> pinfos;
        };
}
```

ProcinfoContentSkin.tmpl:

// For who has experienced various templates solution for Java, PHP and so on, some parts seem pretty familiar

```
<% c++ #include "proclist.h" %>
```
=> We include simply our C++ prototype here

```
<% skin ProcinfoContentSkin %>
```
=> Useful when the template are shared libraries

```
<% view ProcinfoContent uses content::ProcinfoContent %>

<% template render() %>

<html>

        <head>

        <link rel="stylesheet"
href="//jqueryui.com/jquery-wp-content/themes/jquery/css/base.css?v=1
">

        <link rel="stylesheet"
href="//jqueryui.com/jquery-wp-content/themes/jqueryui.com/style.css"
>

        <script src="//code.jquery.com/jquery-1.10.2.js"></script>

        <script
src="//code.jquery.com/ui/1.11.4/jquery-ui.js"></script>

        <script type="text/javascript">

        $(function() {

                        $("tbody").sortable();

                        $("tbody").disableSelection();

        });

 </script>

        </head>
```

```
<body class="jquery-ui page-template-default">
<h1>Processes statistics</h1>
<div class="container">
<div id="content-wrapper">
<div id="content">
<table class="ui-sortable">
<tr>
        <th>PID</th>
        <th>PATH</th>
        <th>ARGUMENTS</th>
        <th>OWNER</th>
</tr>
        <tbody>
        <% foreach info in pinfos %> =>
```

Iterate through the pinfos member of the content's class ...

```
        <% item %>
        <tr> =>
```

... then 'echoing' each field of a Procinfo struct

```
                <td class="ui-state-default ui-sortable-handle"><%= info.pid %></td>
                <td class="ui-state-default ui-sortable-handle"><%= info.pathName %></td>
                <td class="ui-state-default ui-sortable-handle"><%= info.args %></td>
                <td class="ui-state-default ui-sortable-handle"><%= info.userName %> (<%= info.userFullName %>) <%= info.userHome %></td>
```

```
        </tr>
        <% end %>
        <% end %>
        </tbody>
        </table>
        </div>
        </div>
        </div>
        </body>
</html>
<% end template %>
<% end view %>
<% end skin %>
```

4. Application

cppcms_procstat.cc :

```
// And finally the most important, the CppCms's application ...
#include <cppcms/application.h>
#include <cppcms/applications_pool.h>
#include <cppcms/service.h>
#include <cppcms/http_response.h>
```

```
#include <iostream>
#include <sstream>
#include <stdlib.h>


#include <kvm.h>
#include <sys/param.h>
#include <sys/queue.h>
#include <sys/socket.h>
#include <sys/sysctl.h>
#include <sys/types.h>
#include <sys/user.h>


#include <pwd.h>
#include <libprocstat.h>

#include "proclist.h"

class Procstat : public cppcms::application {
private:
        procstat *ps;
        content::ProcinfoContent pc;
public:
        Procstat(cppcms::service &srv) : cppcms::application(srv) {
```

```
        // We're opening the processes info via the internal sysctl system
        // There are other ways, via a kernel's core dump file or via kvm ...
                ps = procstat_open_sysctl();
        }
        ~Procstat() {
                procstat_close(ps);
        }
        virtual void main(std::string url);
};

int
kp_compare(const void *a, const void *b) {
        const kinfo_proc *ka = reinterpret_cast<const kinfo_proc
*>(a);
        const kinfo_proc *kb = reinterpret_cast<const kinfo_proc
*>(b);

        if (ka->ki_pid < kb->ki_pid)
                return -1;
        else
                return 1;
}
```

```
void

Procstat::main(std::string) {

        unsigned int ct;

        int i;

  // we just get the processes information w/o their thread IDS though ...

  // We could get also only a specific group of processes per TTY or user etc ...

        kinfo_proc *kp = procstat_getprocs(ps, KERN_PROC_PROC, 0,
&ct);

        if (kp == NULL)

                return;

        pc.pinfos = std::vector<Procinfo>();

        qsort(kp, ct, sizeof(*kp), kp_compare); // As the processes list is not
ordered, we do per PID


        for (i = 0; i < ct; i ++) {

                char path[PATH_MAX];

                procstat_getpathname(ps, &kp[i], path, sizeof(path));

                if (strlen(path) > 0) {

                        Procinfo pi;

                        pi.pid = kp[i].ki_pid;

                        pi.pathName = std::string(path);


                        std::stringstream ss;

        // Here we get the possible arguments the process were called with ...
```

```
            // args NULL terminated list pointer will be freed by procstat_close later
                                    char **args = procstat_getargv(ps, &kp[i],
0);
                                    char **pargs = args;
            // pargs[0] == path here, so it is bypassed (hence we could have just used
procstat_getargv ...)
                                    while (*++pargs)
                                            ss << " " << *pargs;


                                    pi.args = std::string(ss.str());
                                    passwd pw, *res;
                                    memset(&pw, 0, sizeof(pw));
                                    char buf[1024];
                // Just to get more “human readable” process' user info
                                    if (getpwuid_r(kp[i].ki_ruid, &pw, buf,
sizeof(buf), &res) == 0) {
                                            pi.userName =
std::string(pw.pw_name);
                                            pi.userFullName =
std::string(pw.pw_gecos);
                                            pi.userHome = std::string(pw.pw_dir);
                                    }
pc.pinfos.push_back(pi);
                            }
                    }
```

```
        procstat_freeprocs(ps, kp); // Important to free the processes informa-
tions
        render("ProcinfoContent", pc); // Finally rendering the related template
...
}
int
main(int argc, char *argv[]) {
        try {
                cppcms::service srv(argc, argv);
                srv.applications_pool().mount(
                        cppcms::applications_factory<Procstat>()
                );
        // Now our server is listening client's requests ...
                srv.run();
        } catch (std::exception const &ex) {
                std::cerr << ex.what() << std::endl;
        }

        return 0;
}
```

5. Configuration

CppCms uses the popular JSON format for the configuration's file as follow for our example.

config.json :

```
{
        "service": {
                "api:: "http,",
                "ip: "ip address to listen,",
                "port:": 8180
        },
        "http": {
                "script_names": [ "/procstat" ]
        }
}
```

The possibilities of configuration are pretty rich, here we're using the internal web server but in production, it might be preferable to configure in FastCGI mode and allowing a guenine web server, like Nginx, handling the client's connections.

```
{
        "service": {
                "api:": "fastcgi,
                 socket:": "" <path of the unix socket>,
        },
        "http": {
                "script_names": [ "/procstat" ]
        }
}
```

If we planned to compile the template as a shared library, we would need also to declare it in our config. For more precise information, please read this page.

http://cppcms.com/wikipp/en/page/cppcms_1x_config

6. Compilation

First, we need to “compile” the template file into a C++ code via a CppCms utilily.

```
cppcms_tmpl_cc ProcinfoContentSkin.tmpl -o ProcinfoContentSkin.cc
```

Then compiling altogether our CppCms' application with this template. Indeed, for the sake of the simplicity and as we have only one template, we compile it statically.

```
c++ -g -O2 -I/usr/local/include -L/usr/local/lib -o cppcms_procstat
cppcms_procstat.cc ProcinfoContentSkin.cc -lcppcms -lbooster -lproc-
stat
```

I would advise to use at least a Makefile ... The booster's library is necessary for the template's system otherwise it is also possible to render a HTML content directly in the application's level via an usual C++ stream like here.

```
void

Procstat::main(std::string) {

...

response().out()<<

        "<html>\n<body>\n"

        " <h1>Processes statistics</h1>\n";

...

}
```

7. Test

Once compiled, we can finally launch our CppCms's application.

```
./cppcms_procstat -c config.json
```

*Illustration SEQ "Illustration" *Arabic 1: Here our sortable list of processes*

This is it, we can now read the processes list and rearrange the order in a fancy manner. There is a lot of room for improvements, hopefully, that might give some ideas to you, readers. I hope at least, that will give you also the curiosity to dig in more in the procstat's API.

Development tools on FreeBSD

If you usually program in Linux and you are considering a potential switch to FreeBSD, this article will give you an overview of the possibilities.

1. How to install the dependencies

FreeBSD comes with either applications from binary packages or compiled from sources (ports). They are arranged by software types (programming languages mainly in lang (or java specifically for Java), libraries in devel, web servers in www...) and the main tool for modern FreeBSD versions is pkg, similar to Debian apt tools suite. Hence, most of the time, if you are looking for a specific application/library, simply:

```
pkg search <name>
```

without necessarily knowing the fully qualified name of the package, it is somehow sufficient.

For example:

`pkg search php5` will display php5 itself and the modules, furthermore php56 specific version and so on.

The main difference is, you are not forced to either choose the binary or the port but can have both if it suits your need, but keep in mind that compiling from source can take a certain amount of time to achieve, if that is an important point for you. If the ports tree is not already present on your server, portsnap fetch extract will fetch the ports tree for you by default in /usr/ports. Then, related to the software type described above, you just need to go to the related folder. For example, for installing php5 :

```
cd /usr/ports/lang/php5

make config-recursive

make install clean
```

The second command, depending on which options you are going to choose, will display all the options available for each dependency (for example, if gd support is enabled, furthermore the options for graphics/gd library will appear).

However, most of the time, the binary packages are sufficient to cover most of the needs.

2. Web development

This is basically the easiest area to migrate to. Most Web languages do not use specific platform features, so most of the time, your existing projects might just be “drop-in” use cases.

If your language of choice is PHP, luckily, this scripting language is workable in various operating systems, most of the Unixes and Windows. In the case of FreeBSD, you have even more different ports or binary package versions (5.4 to 5.6). In this particular case, you might need some specific PHP modules enabled, luckily, they are available uatomatically or if the port is the way you chose, it is via the www/php5-extensions one.

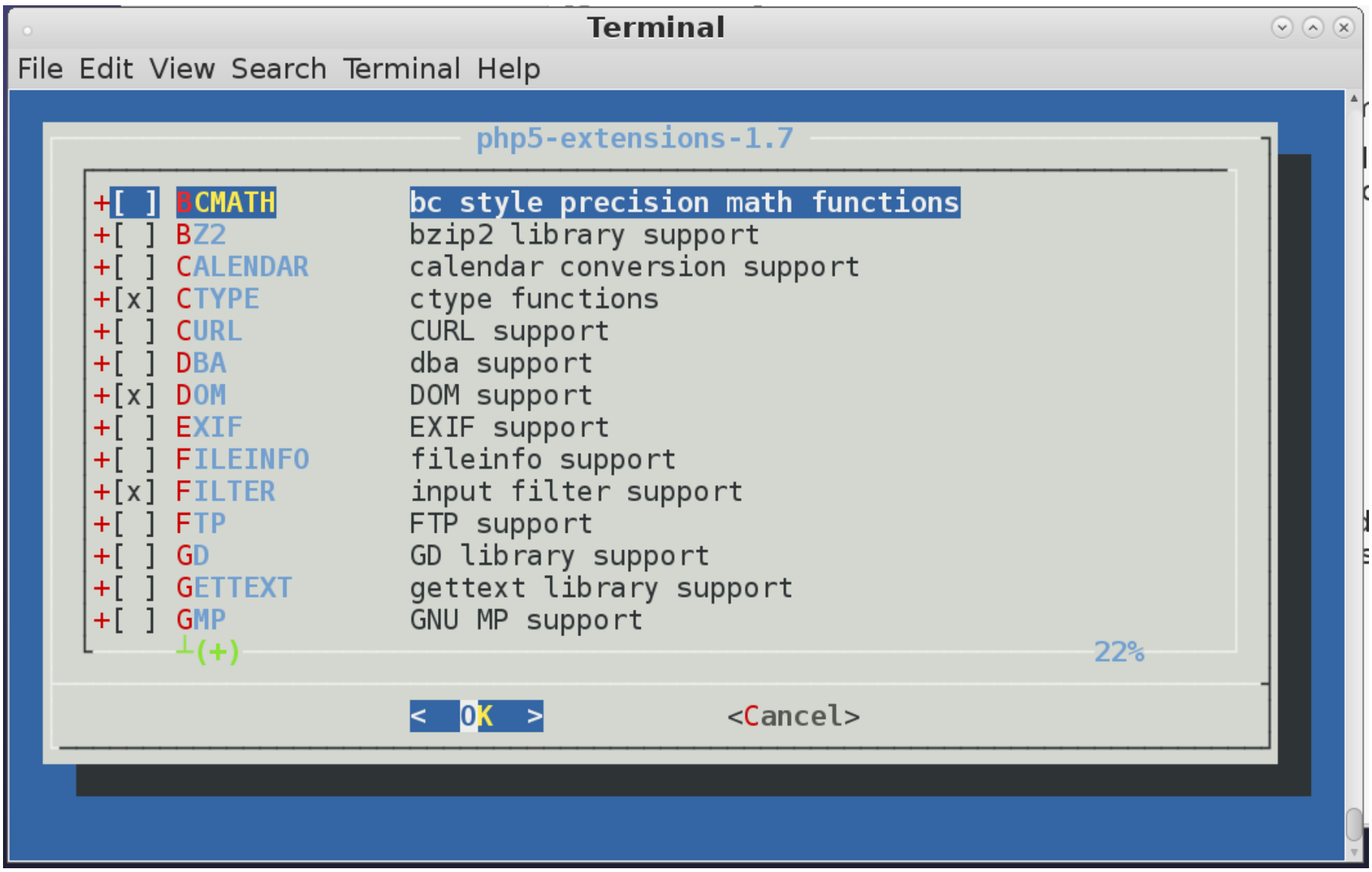

Figure 1. PHP port and modules.

Of course, developing with Apache (both 2.2 and 2.4 series are available, respectively www/apache22 and www/apache24 packages) or even better with Nginx (the last stable or the last development versions could be used, respectively www/nginx and www/nginx-devel packages) via php-fpm is possible.

Outside of PHP, the same applies to Python / Django (www/py-django) and Ruby on Rails (www/rubygen-rails), Python 2.7 and 3.5 (lang/python<version>) are available as Ruby until 2.2 (lang/ruby<version>).

In terms of databases, we have the regular RDBMS like MySQL and PostgreSQL (client and server are distinct packages: `databases/(mysql/portgresql)<version>-client` and `databases/(mysql/postgresql)<version>-server)` and the more modern concept of NoSQL with CouchDB for example `(databases/couchdb)`, MongoDB `(databases/mogodb)`, Cassandra `(databases/cassandra)` to name a few.

Also, if you need to perform efficient Map / Reduce for Big Data work, you have either the well known Apache Hadoop or Apache Spark (respectively devel/hadoop and devel/spark). Last, if you ever need a search engine, Apache Solr/Lucene (`textproc/apache-(solr/lucene)`), Xapian (`databases/xapian`) and their various language bindings are available.

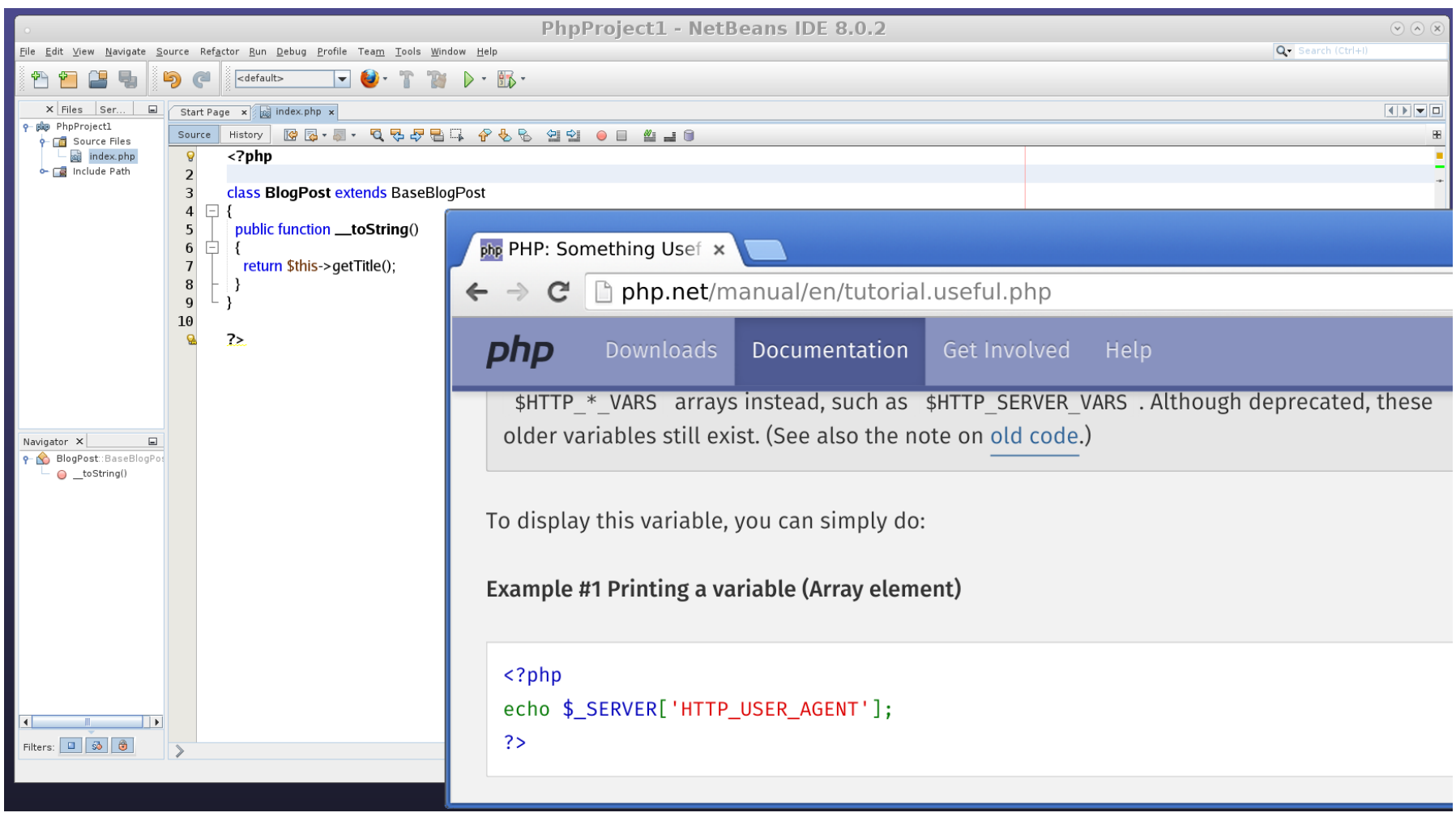

Figure 2. PHP development under Netbeans.

Is it rather Java Web or any language based on the Java VM platform? In FreeBSD, you even have Java 8 (either java/openjdk8 and java/linux-oracle-jdk18), various popular frameworks and J2EE servers or servlet engines like Spring (`java/springframework`), Jboss (`java/jboss<version>`), Tomcat (`www/tomcat<version>`), Jetty (`www/jetty`)... Even the more modern languages like Scala (`lang/scala`), Groovy (`lang/groovy`) can be found.

Two languages described above, Python and Ruby, have their Java VM counterparts, Jython (`lang/jython`) and Jruby (`lang/jruby`), available as well,

In term of Integrated Development Environment, there are still several choices. The venerable Netbeans (`java/netbeans or java/netbeans-devel`), Eclipse (`java/eclipse` ... side note, FreeBSD needs to have Kerberos support enabled, NO_KERBEROS is `/etc/make.conf` or `/etc/src.conf` presence needs to be checked) with their numerous popular plugins.

3. Low level development

The BSD is shipped with a C and C++ compilers in base. In the case of FreeBSD 10.2, it is clang 3.4.1 (in x86 architectures), otherwise modern versions of gcc, for developing with C++11, for example, are of course available too (`lang/gcc<version> ... until gcc 5.2`).

Numerous libraries for various topics are also present, web services SOAP with gsoap through User Interfaces with GTK (`x11-toolkits/gtk<version>`), QT4 or QT 5 (`devel/qt<version>`), malware libraries with Yara (`security/yara`) ...

In term of IDEs, Eclipse and Netbeans described above allow both C/C++ development, Anjuta and Qtcreator are also available for important projects. If you prefer, FreeBSD has in base vi and Vi Improved can be found in ports / packages (editors/vim or editors/vim-lite without X11 support).

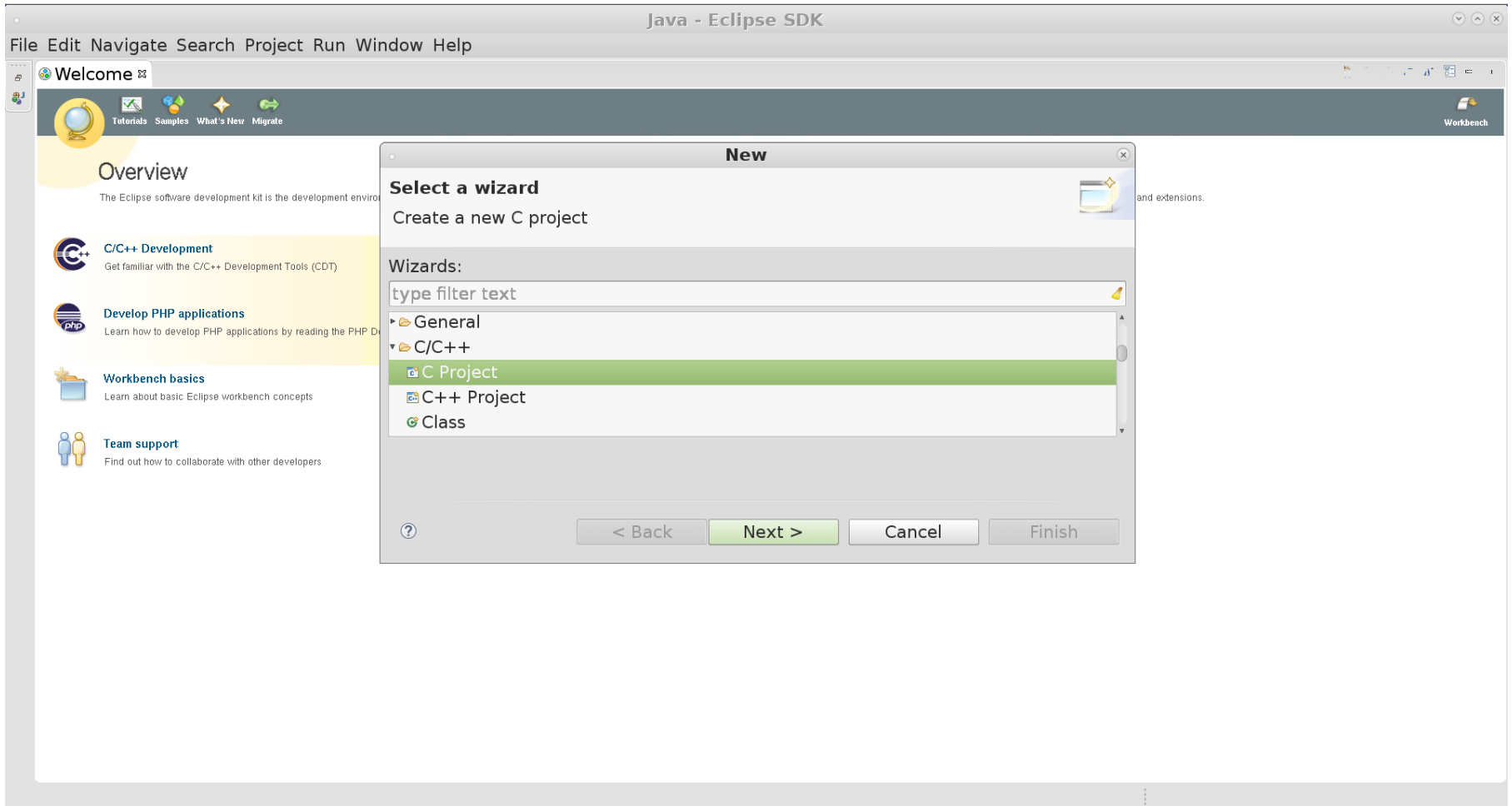

Figure 3. Java Eclipse SDK.

FreeBSD is a POSIX system, hence porting C/C++ code to this platform depends on the degree of portability of your projects, so the usage of specific “linuxisms” and such.

In case more information is needed about porting software in FreeBSD and its specific tools, I would recommend the reading of BSDMag issues no 66 and 68.

4. Android / Mobile development.

In order to be able to do Android development, to a certain degree, the Linux compatibility layer (aka `linuxulator`) needs to be enabled. Also `x11-toolkits/swt` and `linux-f10-gtk2 port/package` need to be installed (note that `libswt-gtk-3550.so` and `libswt-pi-gtk-3550.so` are needed, the current package is versioned as 3557, can be solved with symlinks). In the worst case, remember that `bhyve` (or Virtualbox) are available and can run any Linux distribution smoothly.

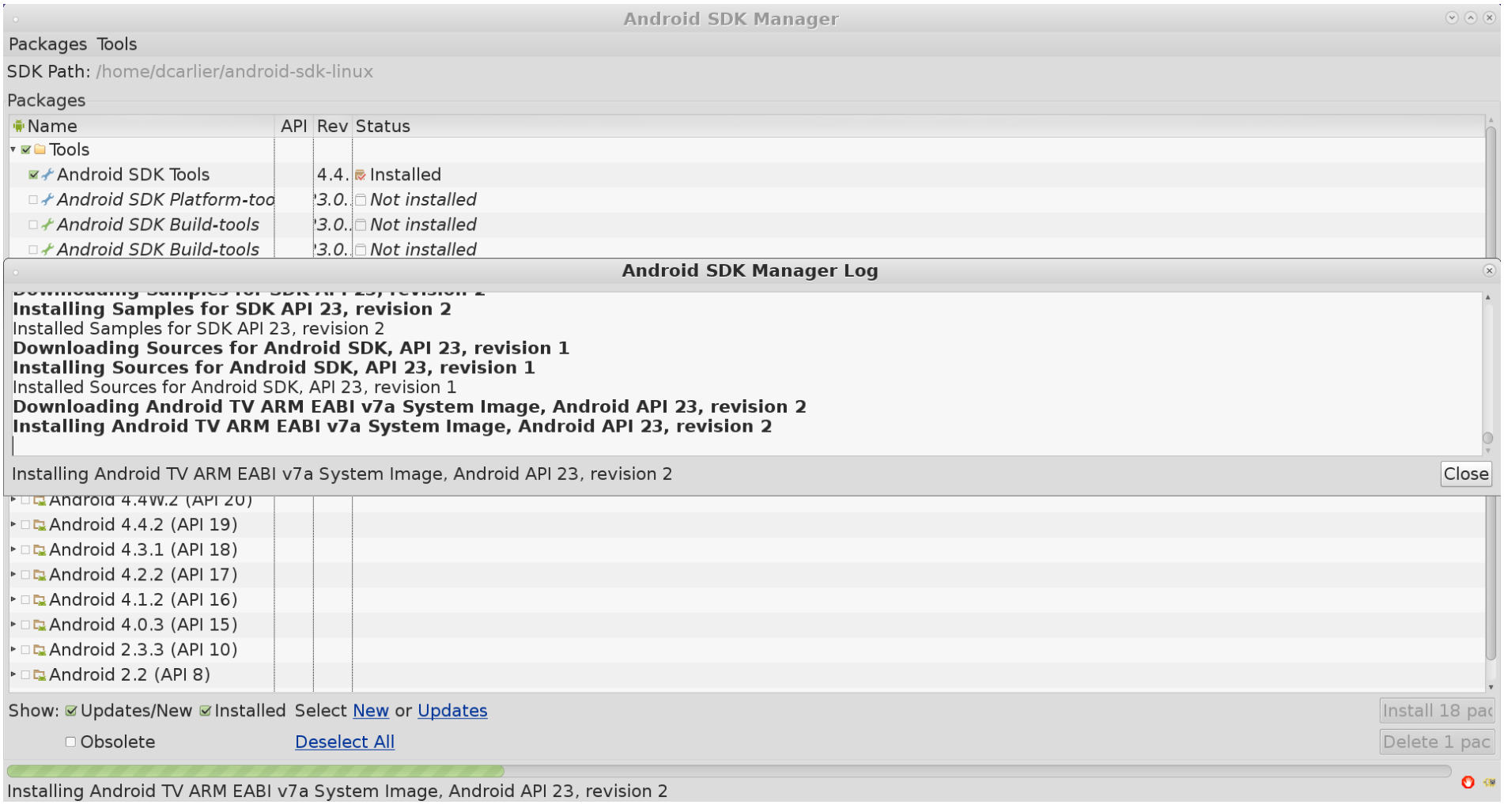

Figure 4. SDK Manager under FreeBSD.

5. Source Control Management.

FreeBSD comes in base with a version of subversion, as FreeBSD source is in a subversion repository, prefixed svnlite, though, to avoid conflicts with the package/port.

In addition, Git is present but via the package/port system with various options (with or without a user interface, subversion support).

6. Conclusion

FreeBSD has made tremendous improvements over the years to fill the gap with Linux whereas it still keeps its own interesting specificities, hence there would not be too many blockers if your projects are reasonably sized to consider a migration to FreeBSD.

The Journey of a C Developer in FreeBSD's World

Moving from Linux to FreeBSD involves quite a number of changes; some gains and some losses. As a developer, for most of the programming languages, especially the high level ones, there are no meaningful disturbing changes. But for languages like C (and its sibling C++), if you want to port your software, libraries, etc., some points might need to be considered.

What you will learn:

- How to move from Linux to FreeBSD
- How to develop under FreeBSD

What you should know:

- Basic knowledge of C programming

The code

As is often the case with C, it is not especially straightforward; the code itself might need some changes, minus the pure POSIX part. Let's say your program needs to use some known network functions.

```
#include <sys/param.h> ⇒ BSD defined, FreeBSD current version etc…

#if defined(BSD)

#include <netinet/in.h>
```

```
#endif

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>


int

main(int argc, char *argv[])

{

    …

    struct in_addr in;

    const char *ip = argv[1];

    if (inet_pton(AF_INET, ip, &in) == -1)

...

}
```

Here we have a more complex case; for example, how do we get the MAC Address of an interface ?

```
int

main(int argc, char *argv[])

{

…

struct ifreq if;

char hwaddr[6] = { 0 };

...
```

```
#if defined(__linux__)
if (ioctl(clsock, SIOCGIFHWADDR, &if) == 0)
    memcpy(hwaddr, if.ifr_hwaddr.sa_data, sizeof(hwaddr));
...
#else if defined(BSD)
struct sockaddr_dl *cl = (struct sockaddr_dl *)(if.ifa_addr);
unsigned char *p = (unsigned char *)LLADDR(cl);
memcpy(hwaddr, p, sizeof(hwaddr));
#endif
…
}
```

In addition, FreeBSD provides a bunch of specific functions, like strlcpy/strlcat (safer versions of strcpy/strcat) and strtonum family functions, all of which are available in the base, whereas Linux must install the separate BSD library to have them. If you have any doubts about any functions, all manpages are available and very well written.

The environment

FreeBSD is shipped by default with clang, whereas Linux relies on GCC suite. If you heavily use OpenMP, clang does not provide it yet so you might need to install GCC from ports. Somehow, clang mostly compiles faster and provides more informative warning and error messages. Fortunately, they share a significant amount of common flags.

On Linux, you may use a custom memory allocator during your development, like jemalloc. It's a very handy and useful library that allows you to generate statistics, to fill freed memory with specific values, and to spot corrupted memory usage.

Good news! You do not need to install it—FreeBSD libc's malloc (aka phkmalloc) uses jemalloc internally. To print statistics from your application, for example, you need to include `malloc_np.h` instead of `jemalloc/jemalloc.h`

As for the makefiles, this is the BSD format which differs from GNU style:

A basic makefile for a library:

```
…
LIB=            mylib
SHLIB_MAJOR=    1
SHLIB_MINOR=    0
```

=> In addition to the static (profiled and non profiled one), it will compile the shared version

```
SRCS=       mylib.c

.include <bsd.lib.mk>
```

A basic makefile for an application:

```
…
PROG=       myprog  => will compile an app called myprog

SRCS=       main.c prog.c
CFLAGS+=        -I${.CURDIR}/../mylib
```

=> always concatenate cflags, some like fstack-protector, -Qunused-arguments … are added automatically

```
LDADD=      -lutil -lmylib
DPADD=      ${LIBUTIL}
```

```
=> linked to libutil.a ${.CURDIR}/../mylib/libmylib.a


.include <bsd.prog.mk>
```

FreeBSD can handle GNU via (gnu)make, libtool, etc. via the ports.

Or to save the effort of porting this part, it might be more handy to use cmake or scons.

The publication

You might want to publish your library / application in pure FreeBSD's path. You can make a port that can provide some options for the user. It can download the source and compile it with its dependencies in a natural manner. In addition, you can build a binary package to facilitate the distribution.

Example of a port Makefile

```
PORTNAME=   mylib

PORTVERSION=    1

PORTREVISION=   0


MAINTAINER= john.doe@email.com


LICENSE=        BSD


OPTIONS_DEFINE=     CURL_SUPPORT

CURL_SUPPORT_DESC= Enable Curl support
```

=> Will display to the user the curl support then will add a flag during compilation

```
if ${PORT_OPTIONS:MCURL_SUPPORT}

CFLAGS+=    -DCURL

.endif



.include <bsd.port.mk>
```

For instance, you can put the archive `.tar.gz` of the library in `/usr/ports/distfiles`, then type make checksum. Then, make install will compile and install it in /`usr/local`. The handbook of making ports is very useful to read.

Furthermore, you can build a binary version of this port to facilitate its distribution. As simple as it is, pkg create myli. It will create a txz archive in the current folder. In the end, `pkg` install mylib will install it.

The conclusion

Developing under FreeBSD is not the extreme challenge you might think it is. Even better, from coding to publishing, everything is thought out and made in a constant way without any external dependencies. If you want to go even further, like kernel development, again it is easy and in base. So there is no real reason to stay away from FreeBSD anymore, you are more than welcome.

About Hardened BSD

The HardenedBSD project was created in 2014 by Oliver Pinter and Shawn Webb. The project aims to provide security enhancements to the FreeBSD project. We plan to upstream most, if not all, of our projects.

The core HardenedBSD team consists of:

- Oliver Pinter
- Shawn Webb

The developer team consists of:

- David Carlier

The FreeBSD Corner

- Nathan Dautenhahn
- Danilo Egea Gondolfo
- Oliver Pinter
- S h a w n W e b b

The following people and organizations have contributed to the HardenedBSD project:

- Ilya Bakulin
- Bryan Drewery
- Danilo Egea Gondolfo
- Dag-Erling Smørgrav
- Robert Watson
- Hunger
- SoldierX - Donated a sparc64 and a BeagleBone Black
- Hyper6 - Designed logo
- Automated Tendencies - Substantial monetary donation.

Kernel and syscalls / Introduction

In this article, we will have an overview of what is called a syscall (system call shortened), from the kernel side to the userland then in the end how to create a new one, in FreeBSD.

It is assumed you know how to build FreeBSD-current and have some knowledge about C language.

1. What is a syscall?

A syscall is a code implemented in the kernel side that can be potentially called from the userland for various purposes, network, generating some random data or controlling the system processes as well. Whatever the type of syscall, the userland never interacts directly with the kernel but via a defined interface. Let's start with a simple example, the `getpagesize` call, which is the number of bytes per page.

```
int getpagesize(void);
```

Let's see how, in the kernel, it is implemented.

```
sys/kern/kern_mib.c

...

SYSCTL_INT(_hw, HW_PAGESIZE, pagesize, CTLFLAG_RD|CTLFLAG_CAPRD,

    SYSCTL_NULL_INT_PTR, PAGE_SIZE, "System memory page size");

...
```

Which simply returns the constant via the sysctl system. Let's see now how it is implemented in the userland side.

```
lib/libc/gen/getpagesize.c

...

int

getpagesize(void)

{

        int mib[2];

        static int value;

        size_t size;

        int error;


        if (value != 0)

                return (value);

/* Here we try to get as much of the cached result as possible */

    error = _elf_aux_info(AT_PAGESZ, &value, sizeof(value));

    if (error == 0 && value != 0)

                return (value);

/* Otherwise we get the result via the sysctl call */

    mib[0] = CTL_HW;

    mib[1] = HW_PAGESIZE;

    size = sizeof value;

    if (sysctl(mib, 2, &value, &size, NULL, 0) == -1)

                return (-1);
```

```
        return (value);
}
...
```

As you can see in this case, mainly the work is done in the userland side. Another type of syscall exists where all the implementation is done in the kernel side only but its symbol is transmitted via a symbol map. Next, let's have an eye on the `getpid syscall`. The usual C function has this signature.

```
pid_t
getpid(void);
```

Those syscalls are organized per type in the kernel, implemented in the files `src/sys/kern/(kern/sys)_*.c`. Basically, the `sys_*` files hold the function implementations but if the complexity requires it, some codes are split in the `kern_*` ones.

First of all, the struct `getpid_args` reflects the userland parameters, every syscall needs a corresponding *`args struct` to pass the userland parameters, hence as `getpid` has none. Here we have a dummy's.

```
src/sys/sys/sysproto.h
...
struct getpid_args {
        register_t dummy;
};
...
```

This type of syscall has this signature:

```
int <syscall>(struct thread *, struct <userland call's name>_args *)
```

The thread parameter is the current thread from which we can get the current process, locking it in order to modify or just read a specific state. The integer returns here to, potentially, set errno in errors code paths.

Here's the corresponding getpid implementation.

In the case of `getpid`, it is always successful, errno is never set so we return 0 directly.

```
src/sys/kern/sys_prot.c

...

int

sys_getpid(struct thread *td, struct getpid_args *uap)

{

        struct proc *p = td->td_proc;


        td->td_retval[0] = p->p_pid;

#if defined(COMPAT_43)

        td->td_retval[1] = kern_getppid(td);

#endif

        return (0);

}

...

int

kern_getppid(struct thread *td)

{

        struct proc *p = td->td_proc;

        struct proc *pp;
```

```
        int ppid;


        PROC_LOCK(p);
    if (!(p->p_flag & P_TRACED)) {
                ppid = p->p_pptr->p_pid;
                PROC_UNLOCK(p);
        } else {
                PROC_UNLOCK(p);
                sx_slock(&proctree_lock);
                pp = proc_realparent(p);
                ppid = pp->p_pid;
                sx_sunlock(&proctree_lock);
        }


        return (ppid);
}
...
/* Returns the real parent process whether it had exited or
traverses the orphaned linked list */
struct proc *
proc_realparent(struct proc *child)
{
        struct proc *p, *parent;
```

```
        sx_assert(&proctree_lock, SX_LOCKED);
        if ((child->p_treeflag & P_TREE_ORPHANED) == 0) {
                if (child->p_oppid == 0 ||
                    child->p_pptr->p_pid == child->p_oppid)
                        parent = child->p_pptr;
                else
                        parent = initproc;
                return (parent);
        }
for (p = child; (p->p_treeflag & P_TREE_FIRST_ORPHAN) == 0;) {
                /* Cannot use LIST_PREV(), since the list head is not known. */
                p = __containerof(p->p_orphan.le_prev, struct proc,
                    p_orphan.le_next);
                KASSERT((p->p_treeflag & P_TREE_ORPHANED) != 0,
                    ("missing P_ORPHAN %p", p));
        }
        parent = __containerof(p->p_orphan.le_prev, struct proc,
            p_orphans.lh_first);
        return (parent);
}
...
```

Now, the next step to see is how the kernel makes `sys_getpid` available to the userland.

Our getpid syscall can be found in `sys/kern/syscalls.master` file

```
20      AUE_GETPID      STD     { pid_t getpid(void); }
```

20 is the syscall identifier, `AUE_GETPID` is the auditing event. Auditing in FreeBSD is the possibility to log different type of informations from those syscalls: authentication, file descriptor operations (via fcntl call), `userid/groupid` changes. In our case, in fact, `AUE_GETPID` is equal to `AUE_NULL`, which means no auditing. At last, STD means it is a standard syscall.

2. How to add your own syscall

FreeBSD allows you to add new syscalls similarly to the ones above.

For the example, let's make a syscall that returns the full command of a given pid (and if the pid is -1 then it would be the current one), which would have this signature:

```
int custom_func(pid_t pid, char *buf, size_t buflen);
```

It might be advised to create a new file instead of updating `sys_generic.c`, let's create `sys_custom.c` inside `sys/kern` folder

```
#include <sys/cdefs.h>

__FBSDID("$FreeBSD$");


#include <sys/param.h>

#include <sys/systm.h>

#include <sys/lock.h>

#include <sys/sx.h>

#include <sys/mutex.h>

#include <sys/proc.h>

#include <sys/errno.h>
```

```
#include <sys/sysctl.h>

#include <sys/sysproto.h>

// Here this is just for the example, having a “optin” to enable our custom function

static int enable_custom = 0;

SYSCTL_INT(_debug, OID_AUTO, enable_custom, CTLFLAG_RW,

        &enable_custom, 0, "Enable custom");


int

sys_custom_func(struct thread *td, struct custom_func_args *uap)

{

        struct proc *p;

        int error = 0;


td->td_retval[0] = 0;

memset(uap->buf, 0, uap->buflen);


if (enable_custom > 0) {

  // We set errno if the buffer is too small or

        if (uap->buflen < MAXCOMLEN) {

                td->td_retval[0] = -1;

                error = EINVAL;

                goto out;

        }
```

```
// if we do not find our specific process
                if (uap->pid > -1) {
                        if ((p = pfind(uap->pid)) == NULL) {
                                td->td_retval[0] = -2;
                                error = ESRCH;
                                goto out;
                        }
                        PROC_UNLOCK(p);
                } else
                        p = td->td_proc;
        // Before copying to the buffer the command, we lock the process
                PROC_LOCK(p);
                strlcpy(uap->buf, p->p_comm, MAXCOMLEN);
                PROC_UNLOCK(p);
        }

out:
        return (error);
}

...
```

To declare our new function, we need to add this in `sys/kern/syscalls.master` file:

```
...

550     AUE_NULL        STD     { int custom_func(pid_t pid, char
*buf, size_t buflen); }
```

Note: the identifier must be unique, we just increment from the previous.

Then, in our FreeBSD source folder, we could type this:

```
make -C sys/kern sysent
```

Now there are new entries in `sys/sys/sysproto.h` header, first we have our userland struct arg:

```
struct custom_func_args {

        char pid_l_[PADL_(pid_t)]; pid_t pid; char pid_r_[PA-
DR_(pid_t)];

        char buf_l_[PADL_(char *)]; char * buf; char buf_r_[PA-
DR_(char *)];

        char buflen_l_[PADL_(size_t)]; size_t buflen; char bu-
flen_r_[PADR_(size_t)];

};
```

and our kernel function declared:

```
int     sys_custom_func(struct thread *, struct custom_func_args *);
```

We need to declare our new file `sys_custom.c` to the `sys/conf/files` file as below:

```
...

kern/sys_custom.c               standard

...
```

So the next time we compile the kernel it will be taken in account. Last, we need to make available this function exporting its symbols:

```
lib/libc/sys/Symbol.map


FBSD_1.4 {

...

    custom_func;

...

}


FBSDprivate_1.0 {

...

    _custom_func;

            __sys_custom_func;

...

}
```

Once we have updated our system, our new function and our new `sysctl` ought to be available.

We could test this new function with this short sample code.

```
int main(int argc, char **argv) {

...

char buf[MAXCOMMLEN];

pid_t pid = getpid();

if (custom_func(pid, buf, sizeof(buf)) == 0)
```

```
printf(“command of pid %d is '%s'\n”, pid, buf);
}
...

% sysctl debug.enable_custom
0
./a.out
<no output>
% sysctl debug.enable_custom=1
0 -> 1
./a.out
command of pid 752 is 'a.out'
```

3. Commiting?

If you think a new syscall is needed then you can either discuss it in the dev freebsd mailing list or proposing a patch. Maintaining locally this kind of code change might bring difficulty when updating the source with new official syscalls being added and conflicting identifiers. At least, hopefully this article gave you the taste to dig into this topic.

FreeBSD Kernel

In this article, we will give an overview of the nature of the FreeBSD's kernel. The important configuration files will be explained in addition to learning how to compile the whole system with more options, with more debugging information. Very useful for kernel development.

What do you need:

- FreeBSD 10.x.
- Machine with at least 4 cores is recommended for the system compilation.
- Genuine hardware or virtualized environment as your convenience.

1. The FreeBSD kernel

FreeBSD, like many kernels, is a monolithic kernel with loadable module support. Hence, it is possible to build FreeBSD's kernel with all needed modules statically or, those ones that support it, as separated dynamic loadable modules.

The latter ones can be loaded and unloaded at will at boot time (`<name of kernel module>_load="YES"` in `/boot/loader.conf` file) or via `kldload/kldunload`.

To have an overview of all currently loaded modules, you can type `kldstat`.

For example, the output looks like the following:

```
Id Refs Address Size Name

1 19 0xffffffff80200000 19ff378 kernel
```

```
2 1 0xffffffff81e11000 4f62 ng_ubt.ko

…

9 1 0xffffffff81e5b000 3bab ng_socket.ko
```

Let's load the DTrace module by typing kldload dtraceall. Then if we type kldstat again, we should see some new entries related to this module:

```
…

10 1 0xffffffff81e5f000 89e dtraceall.ko

11 11 0xffffffff81e60000 9964 opensolaris.ko

12 10 0xffffffff81e6a000 857dba dtrace.ko

...
```

If we add `dtraceall_load="YES"`, we will be able to use Dtrace framework facility. You can find an excellent introduction of dtrace in the BSDmag issue of December 2014:

http://bsdmag.org/download/samba-nfs-and-firewall-new-bsd-issue/

Indeed, Dtrace can be very useful for tracing syscalls.

2. Configuration

In order to build the system, we need the whole source code, hence the kernel and the userland. The userland is simply all the base utilities of FreeBSD. Both kernel and userland code are consistent and tied together, available in the same subversion repository. Apart from pure BSD codes, we can find GNU libraries and software (called contrib code). In addition, for ZFS, CTF (Compact C Type Format debug section, similar to DWARF format but reduced in term of size) and DTrace proper compilations, some CDDL codes are present. Happily, FreeBSD is provided with subversion in base, suffixed distinctly to avoid colliding with the port version.

- Check out the source in `/usr/src` via svnlite
- svnlite co *https://svn0.us-east.freebsd.org/base/stable/10* `/usr/src` (or you can check out the current branch with much newer code but with more instability, you can just replace stable/10 by hed).

To better understand how the kernel options work, let's have a look at `/usr/src/sys/conf/options` file.

```
...
# $FreeBSD$
#
# On the handling of kernel options
#
# All kernel options should be listed in NOTES, with suitable
# descriptions. Negative options (options that make some code not
# compile) should be commented out; LINT (generated from NOTES)
should
# compile as much code as possible. Try to structure option-using
# code so that a single option only switch code on, or only switch
# code off, to make it possible to have a full compile-test. If
# necessary, you can check for COMPILING_LINT to get maximum code
# coverage.
#
# All new options shall also be listed in either "conf/options" or
# "conf/options.<machine>". Options that affect a single source-file
# <xxx>.[c|s] should be directed into "opt_<xxx>.h", while options
# that affect multiple files should either go in "opt_global.h" if
# this is a kernel-wide option (used just about everywhere), or in
# "opt_<option-name-in-lower-case>.h" if it affects only some files.
# Note that the effect of listing only an option without a
```

```
# header-file-name in conf/options (and cousins) is that the last
# convention is followed.
#
# This handling scheme is not yet fully implemented.
#
#
# Format of this file:
# Option name filename
#
# If filename is missing, the default is
# opt_<name-of-option-in-lower-case>.h
AAC_DEBUG opt_aac.h
AACRAID_DEBUG opt_aacraid.h
AHC_ALLOW_MEMIO opt_aic7xxx.h
AHC_TMODE_ENABLE opt_aic7xxx.h
AHC_DUMP_EEPROM opt_aic7xxx.h
AHC_DEBUG opt_aic7xxx.h
AHC_DEBUG_OPTS opt_aic7xxx.h
AHC_REG_PRETTY_PRINT opt_aic7xxx.h
AHD_DEBUG opt_aic79xx.h
AHD_DEBUG_OPTS opt_aic79xx.h
AHD_TMODE_ENABLE opt_aic79xx.h
AHD_REG_PRETTY_PRINT opt_aic79xx.h
```

```
ADW_ALLOW_MEMIO opt_adw.h

…
```

- For each line, the option's name then the file created with the relevant preprocessor defined. If the option is present in your kernel configuration file, let's say AHD_DEBUG_OPTS, it is possible to test if AHD_DEBUG_OPTS is defined and providing some contextual code for this option.
- Let's imagine we did a new shiny kernel module, we could add our proper line in this file. `BSDMAG opt_bsdmag.h`
- Another important file is `/usr/src/sys/conf/file`.

```
...

cam/cam.c optional scbus

cam/cam_compat.c optional scbus

cam/cam_periph.c optional scbus

cam/cam_queue.c optional scbus

cam/cam_sim.c optional scbus

cam/cam_xpt.c optional scbus

cam/ata/ata_all.c optional scbus

cam/ata/ata_xpt.c optional scbus

cam/ata/ata_pmp.c optional scbus

cam/scsi/scsi_xpt.c optional scbus

cam/scsi/scsi_all.c optional scbus

cam/scsi/scsi_cd.c optional cd

cam/scsi/scsi_ch.c optional ch

cam/ata/ata_da.c optional ada | da
```

```
cam/ctl/ctl.c optional ctl

cam/ctl/ctl_backend.c optional ctl

cam/ctl/ctl_backend_block.c optional ctl

cam/ctl/ctl_backend_ramdisk.c optional ctl

cam/ctl/ctl_cmd_table.c optional ctl

cam/ctl/ctl_frontend.c optional ctl

cam/ctl/ctl_frontend_cam_sim.c optional ctl

cam/ctl/ctl_frontend_internal.c optional ctl

cam/ctl/ctl_frontend_iscsi.c optional ctl

cam/ctl/ctl_scsi_all.c optional ctl

cam/ctl/ctl_tpc.c optional ctl

cam/ctl/ctl_tpc_local.c optional ctl

cam/ctl/ctl_error.c optional ctl

cam/ctl/ctl_util.c optional ctl

cam/ctl/scsi_ctl.c optional ctl

cam/scsi/scsi_da.c optional da

cam/scsi/scsi_low.c optional ct | ncv | nsp | stg

cam/scsi/scsi_pass.c optional pass

cam/scsi/scsi_pt.c optional pt

cam/scsi/scsi_sa.c optional sa

...
```

- For each line the relative path to sys, the type of module, if optional it will be compiled with the (lower case) option name written afterwards.
- Again, with our new module, we can add in this file our specific kernel module C file, let's say `workshop_module1.`

```
workshop_bsdmagmodule1.c optional bsdmag
```

3. Build

So now we can create a custom kernel config. Let's call it WORKSHOP.

```
cp /usr/src/sys/<arch>/conf/GENERIC /usr/src/sys/<arch>/conf/WORKSHOP

echo "KERNCONF=WORKSHOP" >> /etc/make.conf
```

(it will pick up the new WORKSHOP configuration file, by default it is the GENERIC one)

Steps to build a system:

First, the `userland` needs to be compiled.

```
go to /usr/src
```

If your machine has multiple cores, it is advised to use them for the system compilation.

It might take several hours depending on your current configuration.

```
make -j<number of cores+1> buildworld (builds the userland)

make -j<number of cores+1> buildkernel (builds the kernel)

make installkernel (install the kernel in /)
```

Possibility to do make `-j<number of cores+1> buildworld kernel` (ie create and install the kernel at once). Restart in single user:

```
go to /usr/src
```

Eventually `mergemaster -p` then make `installworld`:

`mergemaster -FUi => mergemaster` will try to merge various configurations files and asking you how you wish to proceed, merging as possible, replacing with a newer one or keeping the existing.

Restart in normal mode.

You should have now a workable system with the latest fixes/patches for the 10.x branch. But, as a developer, we might need more info from the system for debugging, studying the core dump after a system crash/kernel panic. It is advised, as kernel developer, to enable kernel core dump writing (could be enabled when you installed FreeBSD or, afterwards, can be enabled via dumpdir rc.conf variable) at the cost of disk space consuming (can be potentially important, deleting old ones is necessart). They are, by default, located in /var/crash. In order to debug a kernel crash dump, the kernel compiled with debugging symbols, kernel.debug, is necessary. gdb can already be used this way.

```
Reading symbols from /boot/kernel/ng_ubt.ko.symbols...done.

Loaded symbols for /boot/kernel/ng_ubt.ko.symbols

Reading symbols from /boot/kernel/netgraph.ko.symbols...done.

Loaded symbols for /boot/kernel/netgraph.ko.symbols

Reading symbols from /boot/kernel/ng_hci.ko.symbols...done.

Loaded symbols for /boot/kernel/ng_hci.ko.symbols

Reading symbols from /boot/kernel/ng_bluetooth.ko.symbols...done.

Loaded symbols for /boot/kernel/ng_bluetooth.ko.symbols

Reading symbols from /boot/kernel/ng_l2cap.ko.symbols...done.

Loaded symbols for /boot/kernel/ng_l2cap.ko.symbols

Reading symbols from /boot/kernel/ng_btsocket.ko.symbols...done.

Loaded symbols for /boot/kernel/ng_btsocket.ko.symbols

Reading symbols from /boot/kernel/ng_socket.ko.symbols...done.

Loaded symbols for /boot/kernel/ng_socket.ko.symbols
```

```
Reading symbols from /boot/kernel/dtraceall.ko.symbols...done.
Loaded symbols for /boot/kernel/dtraceall.ko.symbols
Reading symbols from /boot/kernel/opensolaris.ko.symbols...done.
Loaded symbols for /boot/kernel/opensolaris.ko.symbols
Reading symbols from /boot/kernel/dtrace.ko.symbols...done.
Loaded symbols for /boot/kernel/dtrace.ko.symbols
Reading symbols from /boot/kernel/dtmalloc.ko.symbols...done.
Loaded symbols for /boot/kernel/dtmalloc.ko.symbols
Reading symbols from /boot/kernel/dtnfscl.ko.symbols...done.
Loaded symbols for /boot/kernel/dtnfscl.ko.symbols
Reading symbols from /boot/kernel/fbt.ko.symbols...done.
Loaded symbols for /boot/kernel/fbt.ko.symbols
Reading symbols from /boot/kernel/fasttrap.ko.symbols...done.
Loaded symbols for /boot/kernel/fasttrap.ko.symbols
Reading symbols from /boot/kernel/lockstat.ko.symbols...done.
Loaded symbols for /boot/kernel/lockstat.ko.symbols
Reading symbols from /boot/kernel/sdt.ko.symbols...done.
Loaded symbols for /boot/kernel/sdt.ko.symbols
Reading symbols from /boot/kernel/systrace.ko.symbols...done.
Loaded symbols for /boot/kernel/systrace.ko.symbols
Reading symbols from
/boot/kernel/systrace_freebsd32.ko.symbols...done.
Loaded symbols for /boot/kernel/systrace_freebsd32.ko.symbols
Reading symbols from /boot/kernel/profile.ko.symbols...done.
```

```
Loaded symbols for /boot/kernel/profile.ko.symbols

#0 sched_switch (td=0xfffff8011b80a940, newtd=<value optimized out>,
flags=-2123250552) at /usr/src/sys/kern/sched_ule.c:1940

1940 cpuid = PCPU_GET(cpuid);
```

Like the `userland gdb's` counterpart, we can use backtrace (bt).

```
(kgdb) backtrace

#0 sched_switch (td=0xfffff8011b80a940, newtd=<value optimized out>,
flags=-2123250552) at /usr/src/sys/kern/sched_ule.c:1940

#1 0xffffffff8095b139 in mi_switch (flags=Unhandled dwarf expression
opcode 0x93

) at /usr/src/sys/kern/kern_synch.c:492

#2 0xffffffff8099b172 in sleepq_switch (wchan=<value optimized out>,
pri=<value optimized out>) at /usr/src/sys/kern/subr_sleepqueue.c:552

#3 0xffffffff8099afd3 in sleepq_wait (wchan=0xfffff80115316200, pri=U-
nhandled dwarf expression opcode 0x93

) at /usr/src/sys/kern/subr_sleepqueue.c:631

#4 0xffffffff8095aa47 in _sleep (ident=0x0, lock=0xfffff80115316230,
priority=0, wmesg=0xffffffff80ff47f2 "-", sbt=0, pr=0, flags=<value
optimized out>) at /usr/src/sys/kern/kern_synch.c:254

#5 0xffffffff8099f778 in taskqueue_thread_loop (arg=<value optimized
out>) at /usr/src/sys/kern/subr_taskqueue.c:118

#6 0xffffffff8091e234 in fork_exit (callout=0xffffffff8099f6b0
<taskqueue_thread_loop>, arg=0xfffff800078ebe90, fra-
me=0xfffffe0232e9fac0) at /usr/src/sys/kern/kern_fork.c:996

#7 0xffffffff80d4f4fe in fork_trampoline () at
/usr/src/sys/amd64/amd64/exception.S:610
```

```
#8 0x0000000000000000 in ?? ()

Also list if we want to see

(kgdb) list *0xffffffff8095aa47

0xffffffff8095aa47 is in _sleep (/usr/src/sys/kern/kern_synch.c:254).

249 else if (sbt != 0)

250 rval = sleepq_timedwait(ident, pri);

251 else if (catch)

252 rval = sleepq_wait_sig(ident, pri);

253 else {

254 sleepq_wait(ident, pri);

255 rval = 0;

256 }

257 #ifdef KTRACE

258 if (KTRPOINT(td, KTR_CSW))
```

Then, for example, going up in the stack frames calls and so on...

```
(kgdb) up 2

#4 0xffffffff8095aa57 in _sleep (ident=0x0, lock=0xfffff800035c6a30,

priority=0, wmesg=0xffffffff80ff47f2 "-", sbt=0, pr=0,

flags=<value optimized out>) at /usr/src/sys/kern/kern_synch.c:254

254 sleepq_wait(ident, pri);

(kgdb) list

249 else if (sbt != 0)

250 rval = sleepq_timedwait(ident, pri);
```

```
251 else if (catch)
252 rval = sleepq_wait_sig(ident, pri);
253 else {
254 sleepq_wait(ident, pri);
255 rval = 0;
256 }
257 #ifdef KTRACE
258 if (KTRPOINT(td, KTR_CSW))
```

For more information about gdb, a good helpful workshop exists about this topic:

http://bsdmag.org/course/application-debugging-and-troubleshooting-2

Indeed, especially if you run the -CURRENT branch, the kernel can crash for various reasons and this gdb like tool is handy to have a basic understanding of the reasons...

4. Detecting the potential deadlocks.

Indeed, FreeBSD does not rely on the Giant Lock model anymore, it is based on fine grained level process locking/unlocking. Hence the resulting programming can be tricky and it is easy to get lock contentions.

With this workshop module, you learned the basics of kernel custom configuration, compiling the whole system.

---

kgdb

GNU gdb 6.1.1 [FreeBSD]

Copyright 2004 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

The FreeBSD Corner

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "amd64-marcel-freebsd"...

NetBSD and pkgsrc-wip*

For this mid-summer, we will approach a lighter subject, NetBSD and its ports system. Pkgsrc is the framework to build third party packages for this system. We will see how to create a package and hopefully submit it. Hence, the pkgsrc is supposedly already in your system. Otherwise, a full guide is available here.

1. Environment

It is recommended to install pkglint which will serve to produce a better package. Indeed, as its suffix suggests (lint, the historical C code analyser), it will check the whole package structure, the Makefile, the checksum and so on.

Secondly, you need to choose a main category for your library or application, even if your future package can possibly recover several. For the article, we will choose security/yara, the popular malware searcher library, as an example.

2. Makefile

```
# $NetBSD: Makefile,v 1.2 2015/06/06 08:57:18 pettai Exp $
```

=> This comment is mandatory but when you create for the first time the package it's simply

```
# $NetBSD$>
```

```
PKGNAME=    yara-${YAVER}
```
=> The name of the package and its version

```
CATEGORIES=security
```
=> Its categories, can have several

```
COMMENT=        Pattern matching swiss knife for malware researchers
```

=> Describes briefly the package, more explanations in DESCR file

`WRKSRC= ${WRKDIR}/yara-${YAVER}` => WRKDIR represents where the source port will be extracted (generally it is work/<package name>-<version>)

`USE_TOOLS+=pkg-config automake autoreconf` => Necessary tools to build the package. Could be cmake, perl. They will be installed if not present

`USE_LIBTOOL= yes`

`GNU_CONFIGURE= yes` => Uses GNU version of configure script

```
PKGCONFIG_OVERRIDE+=  libyara/yara.pc.in
```

```
pre-configure:
    cd ${WRKSRC} && autoreconf -fiv
```

=> We can override many sub tasks, related to different steps, before, after the archive extraction, configure, build, installation and so on

`.include "../../security/yara/Makefile.common"` => Makefile.common is used by at least two packages (in our case py-yara) and it regroups common information, could be the dependencies, the version ...

`.include "../../mk/bsd.pkg.mk"` => Mandatory file to include, it contains the main necessary variables

Now, let's have a look at the Makefile.common

```
# $NetBSD: Makefile.common,v 1.3 2015/06/14 21:28:44 pettai Exp $
#
# used by security/yara/Makefile
# used by security/py-yara/Makefile
```

`DISTNAME= v3.3.0` => In case the archive does not have the same name as the package when it is downloaded from the MASTER_SITES set below, this variable needs to be set

`YAVER= ${DISTNAME:S/v//}` => Simply defining the version, in this case we just subtract the v prefix

`MASTER_SITES= ${MASTER_SITE_GITHUB:=plusvic/yara/archive/}` => Some predefined popular URLs like github here, or Sourceforge through predefined variables, hence we just need to give the rest

`DIST_SUBDIR= yara`

`MAINTAINER=`*`pettai@NetBSD.org`*

`HOMEPAGE=` *`https://plusvic.github.io/yara/`*

`LICENSE= apache-2.0` => Likewise, it exists with some predefined licenses, 2 clause BSD, different flavors of GPL ... or we can define a custom one, a simple text file to place inside the licenses subfolder then the user will need to add in its ACCEPTABLE_LICENSES environment variable, hence accepting explicitly this license in order to build the package

3. DESCR and PLIST

We talked earlier about the DESCR file, it is simply a text file which describes more completely the package in question like below.

YARA is a tool aimed at (but not limited to) helping malware researchers to identify and classify malware samples. With YARA you can create descriptions of malware families (or whatever you want to describe) based on textual or binary patterns.

We also need to know the list of files to be (un)installed relative to the variable PREFIX (usually `/usr/pkg`). It is the role of the PLIST file.

```
@comment $NetBSD: PLIST,v 1.1 2015/06/06 08:18:17 pettai Exp $
```

```
bin/yara

bin/yarac

include/yara.h

include/yara/ahocorasick.h

include/yara/arena.h

include/yara/atoms.h

include/yara/compiler.h

include/yara/error.h

include/yara/exec.h

include/yara/filemap.h

include/yara/hash.h

include/yara/libyara.h

include/yara/limits.h

include/yara/modules.h

include/yara/object.h

include/yara/re.h

include/yara/rules.h

include/yara/scan.h

include/yara/sizedstr.h

include/yara/strutils.h

include/yara/types.h

include/yara/utils.h

lib/libyara.la
```

```
lib/pkgconfig/yara.pc

man/man1/yara.1

man/man1/yarac.1
```

4. Patches

Sometimes, the software in question needs to be patched in order to work properly. The patches subfolder should contain the necessary diff files, by convention named patch-<path to the file, dashes replaces by underscores>. In our case, we have `patch-libyara_proc.c` which just needs to add NetBSD support. The patchset is created via make patches.

```
$NetBSD: patch-libyara_proc.c,v 1.1 2015/06/06 08:18:17 pettai Exp $

Add NetBSD support

--- libyara/proc.c.orig   2015-06-06 06:50:32.000000000 +0000
+++ libyara/proc.c
@@ -153,7 +153,7 @@ int yr_process_get_memory(
 #include <yara/mem.h>

 #if defined(__FreeBSD__) || defined(__FreeBSD_kernel__) || \
-    defined(__OpenBSD__) || defined(__MACH__)
+    defined(__OpenBSD__) || defined(__MACH__) || defined(__NetBSD__)
 #define PTRACE_ATTACH PT_ATTACH
 #define PTRACE_DETACH PT_DETACH
 #endif
```

5. buildlink3.mk

Eventually, if it's a library we can create the `buildlink3.mk` file, if another package needs yara library as a dependency, this package just needs to include this file:

```
# $NetBSD: buildlink3.mk,v 1.2 2015/06/06 08:57:18 pettai Exp $

BUILDLINK_TREE+=   yara

.if !defined(YARA_BUILDLINK3_MK)
YARA_BUILDLINK3_MK:=

BUILDLINK_API_DEPENDS.yara+=  yara>=3.3.0
BUILDLINK_PKGSRCDIR.yara?=../../security/yara
.endif  # YARA_BUILDLINK3_MK

BUILDLINK_TREE+=   -yara
```

6. distinfo

Once we have all the pieces needed, we can finally create our distinfo file which stores the checksums of the DISTFILES and eventually the patches. It is created, ideally, via make makesum.

```
$NetBSD: distinfo,v 1.2 2015/06/14 21:28:44 pettai Exp $

SHA1 (yara/v3.3.0.tar.gz) = 6f72d80f21336c098f9013212d496d3920d9ef18
RMD160 (yara/v3.3.0.tar.gz) =
330de9de9294953a3a42032ccc5ae849f065ab5e
```

```
Size (yara/v3.3.0.tar.gz) = 7634474 bytes

SHA1 (patch-libyara_proc.c) =
b860701d604276c8ccd7596f63aa0d02d01a39bc
```

7. Checking the package

`pkglint` will display every part of the package which is not correct, the FATAL messages must be taken into account, some WARNING messages, too.

```
> pkglint

looks fine. => Ideal, but a correct package can have few harmless warnings too.
```

8. Submit

There is a project which aims to get more people involved in investing their time to create packages for pkgsrc. It is called pkgsrc-wip and can be found here *http://pkgsrc-wip.sourceforge.net* and if your package is correct enough you can get commiter bit. I hope this article gave you the taste to create yours.

HardenedBSD, always ahead in security

Previously, I focused on the HardenedBSD project, handled by Oliver Pinter and Shawn Webb, especially the Address Space Randomization Layout feature. The HardenedBSD, also has other features available and I'll try to describe all the features.

1. arc4Random and Chacha 20

Currently, FreeBSD uses the RC4 stream cipher for the arc4random family functions, both in kernel and userland side. These functions serve many purposes, for example, in the kernel side; it allows the creation of proper randomized processes id, the stack protection canaries, and the HardenedBSD Address Space Randomization Layout uses it as well.

In the userland side; openssh uses it widely and is also used in the stack protection counterpart. It is generally an important piece of software.

Recently in the last Hackfest (and previously in the last EuroBSDCon), Theo de Raadt discussed the arc4random OpenBSD's version and raised the need to move on from RC4 to a stronger stream cipher. Hence, the invention of Chacha 20, implemented after the 5.5 release.

Subsequently, we decided to update the HardenedBSD as well, in both kernel and userland side. In the kernel side, the challenge was to keep it SMP safe while keeping the code change smooth and wise while in the userland side, the challenge was to update the fork detection. Indeed when a fork is created, the reseeding is triggered. Usually, getpid function is used for this purpose but we thought there might be a better and more solid approach. M. Dempsey, an OpenBSD contributor, provided a new minherit flag, `MAP_INHERIT_ZERO` to ensure that the memory map is properly zeroed in this case. So, for HardenedBSD, a new `INHERIT_ZERO` flag was added.

Related to this, a new system called, getentropy was added as well. Basically, it fills a buffer of randomized bytes with maximum of 256 bytes. It serves more as an initial input for randomization rather than using it directly. Hence, for example, it can replace a couple of `sysctl/KERN_*RND` calls.

```
#include <unistd.h>

#include <err.h>


int

main(int argc, char *argv[])

{

    char buf[256];

    // errno can be set to EFAULT

    // or EIO (if more than 256 bytes are attempted)

    if (getentropy(buf, sizeof(buf)) != 0)

        errx(1, "getentropy failed");

                …

    return (0);

}
```

2. Some other libc functions

Again, we got inspired by OpenBSD and added some of their useful libc functions.

`getdtablecount` gives the number of file descriptors per process. It can be helpful alongside `getdtablesize`.

```
#include <unistd.h>

#include <err.h>
```

HardnedBSD

```
#define FDRESERVE  5


int

main(int argc, char *argv[])

{

    …

    if (getdtablesize() - getdtablecount() < FDRESERVE)

        errx(1, “running out of file descriptors”);

    ...

}
```

- **reallocarray** checks some potential overflows (but does not zerofy)

```
#include <stdlib.h>

#include <err.h>


int

main(int argc, char *argv[])

{

int *p, *q;

    // i.e same as realloc(NULL, 2 * sizeof(*p)); ...

    p = reallocarray(NULL, 2, sizeof(*p));

    if (p == NULL)
```

```
errx(1, “reallocarray 1 failed”);
    …
    q = reallocarray(p, 10, sizeof(*q));
    if (q == NULL) {
        free(p);
        p = NULL;
        errx(1, “reallocarray 2 failed”);
}
p = q;
    ...
}
```

A slightly different version of strlcpy is provided. strlcpy usually guarantees a zero at the end of the buffer. But the buffer does not sanitize the potential remaining bytes. So our version combines both strlcpy and strncpy advantages at the cost of a slight performance hit, only HardenedBSD, at the moment, does it.

```
#include <string.h>


int
main(int argc, char *argv[])
{
```

```
char buf[10];
    // Will zerofy all the remaining bytes after the first three.
    strlcpy(buf, “foo”, sizeof(buf));
    ...
}
```

For the last, the crypt API was updated recently. Two new functions were added, `crypt_newhash` and `crypt_checkpass`. The latter provides an easy interface to test the validity of a password, while the first allows the creation of a hashed password. Once again, inspired by OpenBSD.

```
#include <crypt.h>
#include <err.h>

int
main(int argc, char *argv[])
{
    const char *passwd = argv[1];
    char hash[_PASSWORD_LEN];
    …
    // errno can be set to EINVAL
    // Second parameter is the hash algorithm preference
    // the default is set if NULL
    if (crypt_newhash(passwd, NULL, hash, sizeof(hash)) != 0)
        errx(1, “crypt_newhash failed”);
```

```
…
    // errno can be set to EACCES
    if (crypt_checkpass(passwd, hash, sizeof(hash)) != 0)
        errx(1, “crypt_checkpass failed”);
    ...
}
```

A secure webserver on FreeBSD with Hiawatha

In most cases, when it comes to choosing a web server, Nginx comes quickly to mind (I personally appreciate this one a lot, no doubt about this). However, an interesting alternative exists that embeds some very nice features, an alternative called Hiawatha.

1. Features

What makes Hiawatha special? First, its code is well audited and famous for its solidness in term of security. Apart from having CGI/FastCGI support (hence possibly making dynamic website with PHP-fpm), SSL, Ipv6, Virtual hosts. it also provides a protection against SQL injection and XSS, CSRF natively. It might lack third party modules support, as the architecture does not allow it but. Hiawatha has Reverse Proxy support!

Luckily, FreeBSD already has a package / port. So once installed.

2. Configuration

Let's configure it. Here's a sample configuration, a FastCGI's one. You can see that it appears very human readable.

```
ErrorHandler = 404:/404.html


Binding {
        Port = 80
        Interface = ::1
        MaxKeepAlive = 30
```

Security

```
   # Two first secure measures, limiting the client request size and the max time for
   # the maximum time for a client's connection kept opened
        MaxRequestSize = 512
          TimeForRequest = 3,20
}
...
Binding {
          Port = 443
          Interface = ::1
          ...
          SSLcertfile = hiawata.pem
          RequireTLS = yes
          # Add X-Random header with a 256 value long
          RandomHeader = 256
}
...
# A feature to make decisions based on url regexes
UrlToolkit {
          ToolkitID = phprewrite
           Method GET
           Match /private DenyAccess
           Match ^/page/(.*) Rewrite /index.php?page=$1
}
```

Security

```
FastCGIServer {

        FastCGIid = PHP5

        UseToolkit = phprewrite

      # Can be the path to the unix socket too

        ConnectTo = 127.0.0.1:2005

        Extension = php

}


VirtualHost {

    Hostname = www.example.com

            WebsiteRoot = /usr/local/www/hiawatha

            AccessLogfile = /var/log/hiowatha/example-access.log

            ErrorLogfile = /var/log/hiowatha/example-error.log

           # Althought those directives can provide protection, they are of course
# not 100% reliable

            PreventCSRF = yes

            PreventSQLI = yes

            PreventXSS = yes

           # A body which matches this regex is forbidden

            DenyBody = ^.*%3Cscript.*%3C%2Fscript%3E.*$

            UseFastCGI = PHP5

            TimeForCGI = 5

            # By default it is index.html
```

```
            StartFile = index.php

}
```

So, for the moment, nothing special and never seen before. Let's dig in more ... Indeed, with Hiawatha, we can set some banning policies like below:

```
# If in our VirtualHost above a user had the bad body content

BanOnDeniedBody = 300

# If too much malformed HTTP requests are made by a client, it's banned for 1 min.

BanOnGarbage = 60

# Banned if the request size exceeds this size

BanOnMaxReqSize = 512

# Anti flood measure, here if the client does more than 20 requests per second, it's banned for
1 min.

BadOnFlooding = 20/1:60

BanOnSQLI = 60

KickonBan = yes

# Reset the ban times if the client attempts to connect when it is already banned

RebanDuringBan = yes

GarbageLogFile = /var/log/hiawatha/garbage.log simple
```

Also Hiawatha provides a separated feature, called Monitor, a feature a bit “a la” Munin but well adapted to this server.

```
# This IP address will be allowed to download the event log files for its analytics

MonitorServer = 192.168.1.2
```

As mentioned before, Hiawatha does not support third party modules, so if you wish for your service to make a decision based on country per IP with geoip, for example, you can use Hiawatha as a Reverse Proxy in front of Nginx. It's as simply as it is shown below inside your VirtualHost setting.

```
VirtualHost {

        Hostname = www.myreverseproxy.com

        PreventCSRF = yes

        PreventSQLI = yes

        PreventXSS = yes

        ReverseProxy .* http://192.168.1.2:8111/geoip

}


# Cache internally those contents from reverse proxy requests per extension

CacheRProxyExtensions = gif,png,jpeg,css

CacheSize = 256

...
```

3. Conclusion

Hopefully, this article will give you the curiosity to consider this approach. For what it provides, it is impressive regarding the fact that it is a one man work done since 2002 (by Hugo Leisink).

NodeJS and FreeBSD - Part 1

Nodejs is well known to allow building server applications in full JavaScript.

In this article, we'll see how to build nodejs from source code on FreeBSD. You will need autoconf tools, GNU make, Python, linprocfs enabled and libexecinfo installed. GCC/ G++ compiler suite (C++11 compliant, ideally 4.8 series or above) or possibly clang can be used to compile the whole source.

To start, we need the nodejs source code from this url *http://www.nodejs.org/dist/latest* where we can find this archive (during the article writing, the last version known is 0.12.2), `node-v<version>.tar.gz`.

Be prepared to be patient, you have enough time for a cup of coffee, the compilation time needed can be quite long...

Once downloaded and extracted, the famous command trio needs to be typed:

- `./configure --dest-os=freebsd`
- `gmake`
- `gmake install`

It's pretty straightforward on first glance. On FreeBSD, when v8 is compiled we get some compilation errors:

```
clang++ '-DV8_TARGET_ARCH_X64' '-DENABLE_DISASSEMBLER' '-DENABLE_HAN-
DLE_ZAPPING' -I../deps/v8  -pthread -Wall -Wextra -Wno-unused-
parameter -m64 -fno-strict-aliasing -I/usr/local/include -O3
-ffunction-sections -fdata-sections -fno-omit-frame-pointer -fdata-
sections -ffunction-sections -O3 -fno-rtti -fno-exceptions -MMD -MF
```

NodeJS

```
/root/node-v0.12.2/out/Release/.deps//root/node-v0.12.2/out/Release/o
bj.target/v8_libbase/deps/v8/src/base/platform/platform-freebsd.o.d.r
aw  -c -o
/root/node-v0.12.2/out/Release/obj.target/v8_libbase/deps/v8/src/base
/platform/platform-freebsd.o
../deps/v8/src/base/platform/platform-freebsd.cc

../deps/v8/src/base/platform/platform-freebsd.cc:159:11: error: mem-
ber reference base type 'int' is not a structure or union

    result.push_back(SharedLibraryAddress(start_of_path, start,
end));

    ~~~~~~^~~~~~~~~~

../deps/v8/src/base/platform/platform-freebsd.cc:191:53: error: use
of undeclared identifier 'MAP_NORESERVE'

                        MAP_PRIVATE | MAP_ANON | MAP_NORESERVE,

                                                ^

../deps/v8/src/base/platform/platform-freebsd.cc:263:48: error: use
of undeclared identifier 'MAP_NORESERVE'

                    MAP_PRIVATE | MAP_ANON | MAP_NORESERVE,

                                             ^

../deps/v8/src/base/platform/platform-freebsd.cc:291:40: error: use
of undeclared identifier 'MAP_NORESERVE'

            MAP_PRIVATE | MAP_ANON | MAP_NORESERVE | MAP_FIXED,

                                     ^

4 errors generated.
```

Ok, so a result variable ought to be a std::vector but it’s considered wrongly as an int and furthermore a wrong mmap flag is used. Let’s fix it!

```
std::vector<SharedLibraryAddress> result;
  static const int MAP_LENGTH = 1024;
  int fd = open("/proc/self/maps", O_RDONLY);
  if (fd < 0) return result;
  while (true) {
    char addr_buffer[11];
    addr_buffer[0] = '0';
    addr_buffer[1] = 'x';
    addr_buffer[10] = 0;
    int result = read(fd, addr_buffer + 2, 8);
    if (result < 8) break;
    unsigned start = StringToLong(addr_buffer);
    result = read(fd, addr_buffer + 2, 1);
    if (result < 1) break;
    if (addr_buffer[2] != '-') break;
    result = read(fd, addr_buffer + 2, 8);
    if (result < 8) break;
    unsigned end = StringToLong(addr_buffer);
    char buffer[MAP_LENGTH];
    int bytes_read = -1;
  do {
     bytes_read++;
     if (bytes_read >= MAP_LENGTH - 1)
```

```
      break;
    result = read(fd, buffer + bytes_read, 1);
```

Apparently, there are two different variables with the same name. Let's rename the second, the int type, to res, for example so the vector result variable can legitimately call push_back method. That fixes the first error.

```
    std::vector<SharedLibraryAddress> result;
  static const int MAP_LENGTH = 1024;
  int fd = open("/proc/self/maps", O_RDONLY);
  if (fd < 0) return result;
  while (true) {
    char addr_buffer[11];
    addr_buffer[0] = '0';
    addr_buffer[1] = 'x';
    addr_buffer[10] = 0;
    int res= read(fd, addr_buffer + 2, 8);
    if (res < 8) break;
    unsigned start = StringToLong(addr_buffer);
    res = read(fd, addr_buffer + 2, 1);
    if (res < 1) break;
    if (addr_buffer[2] != '-') break;
    res = read(fd, addr_buffer + 2, 8);
    if (res < 8) break;
```

```
  unsigned end = StringToLong(addr_buffer);
  char buffer[MAP_LENGTH];
  int bytes_read = -1;
do {
    bytes_read++;
    if (bytes_read >= MAP_LENGTH - 1)
      break;
    res = read(fd, buffer + bytes_read, 1);
```

Let's have a look at the mmap problem.

`MAP_NORESERVE` is a specific flag which guarantees no swap space will be used for the mapping. However, it is a flag usable on Linux and Solaris `/SunOS`.

```
mmap(OS::GetRandomMmapAddr(),
                        size,
                        PROT_NONE,
                        MAP_PRIVATE | MAP_ANON | MAP_NORESERVE,
                        kMmapFd,
                        kMmapFdOffset);
=>
mmap(OS::GetRandomMmapAddr(),
                        size,
                        PROT_NONE,
                        MAP_PRIVATE | MAP_ANON,
```

```
                        kMmapFd,
                        kMmapFdOffset);
void* reservation = mmap(OS::GetRandomMmapAddr(),
                              request_size,
                              PROT_NONE,
                              MAP_PRIVATE | MAP_ANON | MAP_NORESERVE,
                              kMmapFd,
                              kMmapFdOffset);
=>

void* reservation = mmap(OS::GetRandomMmapAddr(),
                              request_size,
                              PROT_NONE,
                              MAP_PRIVATE | MAP_ANON,
                              kMmapFd,
                              kMmapFdOffset);
```

Once modified in every mmap call, we can now retry compiling. However, we get another compilation error. This time, it casts a `pthread_self` returns call to an int.

```
deps/v8/src/base/platform/platform-posix.cc:331:10: error:
static_cast from 'pthread_t' (aka 'pthread *') to 'int' is not al-
lowed

  return static_cast<int>(pthread_self());
```

The problem is, on FreeBSD, a pthread_t type is not an integral type at all but an opaque struct.

Instead, we might replace this line by:

```
return static_cast<int>(reinterpret_cast<inptr_t>(pthread_self()));
```

Now we are finally able to compile. After a couple of minutes, it is finished but we have still one source to update: `lib/dns.js`. Add these two lines after line 127:

```
if (process.platform === 'freebsd' && family !== 6)

        hints &= ~exports.V4MAPPED;
```

Because FreeBSD does not support this flag, it ought to be cleared.

This is all for compilation and it is ready to be used. Next time, we'll have an overlook in the application's building part and ought to see the potential of this library.

NodeJS and FreeBSD - Part 2

Previously, we've seen how to build NodeJS from the sources in FreeBSD with minor source code changes. This time, we'll have an overview of the application's build process.

Numerous excellent tutorials exist to build a nodejs' application in pure Javascript. However, the possibility also exists to build an application natively in C/C++. It is exactly what we're going to see.

1. NodeJs application structure

We only focus on the modern way to build a native application. Before, we had to do a node-waf package via a Python script. It is deprecated and was replaced by node-gyp. This is a basic gyp project structure:

```
<project folder>

--> binding.gyp

--> <C++ source code>
```

A binding.gyp file describes the source code to compile, the package name, eventually the necessary compilation/linker flags. Let's start with the usual Hello world example, quite FreeBSD.

2. Hello world

First, we need an entry point, an initializer, from which we will export our functions to nodejs.

```
void Init(Handle<v8::Object> exports)

{

}
```

And to register our module.

```
NODE_MODULE(freebsdmod, Init)
```
=> Note that there is no need of a comma after this macro

Very well, but for the moment our module is not useful yet, we would need at least one feature.

Let's imagine a simple random function that uses, internally, one of our arc4random family functions, a function that will be called from a nodejs script. The signature of this function would be:

```
void Random(const v8::FunctionCallbackInfo<v8::Value> &);
```

We can imagine that, from the nodejs script, we would like to provide a max value limit as a unique argument:

```
#include <stdlib.h>

#include <node.h> => includes both node and v8 structures ...


using namespace v8;


void Random(const FunctionCallbackInfo<Value> &args){

    Isolate *isolate
```
= Isolate::GetCurrent(); => Here, we get the current v8 engine instance
```
    unsigned long value = 0;

if (args.Length() != 1)

                isolate->ThrowException(Exception::TypeError(

                 String::NewFromUtf8(isolate, "Needs an argument")));

        if (args[0]->IsNumber())
```
=> the arguments are conveniently wrapped, we have access to the caller arguments.

```
value = static_cast<unsigned long>(argc4random_uniform(args[0]-
>NumberValue());

       else

           isolate->ThrowException(Exception::TypeError(

               String::NewFromUtf8(isolate, "The argument is not a
number")));


       args.GetReturnValue().Set(Number::New(isolate, value));

}

void Init(Handle<Object> exports)

{

   NODE_SET_METHOD(exports, "random", Random); => We finally export
our Random function here

}
```

Now, let's have a look a the `binding.gyp` file.

```
{

   "targets": [

   {

      "target_name": "freebsdmod", => represents the name of our mod-
ule

      "sources": ["freebsdmod.cc"]

   }

   ]

}
```

Simply, as it is, it is sufficient for this first example. Now, we can compile our module.

```
> node-gyp configure

> node-gyp build
```

We can now test with a simple nodejs script.

```
var fmod = require('./build/Release/freebsdmod');

var rnd = fmod.random((1024 * 1024));

console.log(rnd); => Should print a significant numerical value
```

3. Wrapped objects

Apart from making atomic C++ functions to export, we also have the possibility to handle more complex cases, by making wrapped node objects. For this example, let's use yara library, the malware's tool. The binding.gyp file would look like this:

```
{
   "targets": [
   {
      "target_name": "yaranode",
      "sources": ["yaranode.cc"],
      "include_dirs":["/usr/local/include"],
      "libraries": ["-L/usr/local/lib", "-lyara"]
   }
   ]
}
```

NodeJS

A wrapped object must inherit ObjectWrap class.

```
#ifndef YARANODE_H

#define YARANODE_H


#include <yara.h>


#include <node.h>

#include <node_object_wrap.h>



static void addrulecb(int, const char *, int, const char *, void *);


class YaraNode : public node::ObjectWrap {

private:

    YR_COMPILER *yc;

    int yrrules;

    explicit YaraNode();

    ~YaraNode();


    static void New(const v8::FunctionCallbackInfo<v8::Value>&);

    static v8::Persistent<v8::Function> constructor;
```

=> Contrary to the Local handles, a Persistent storage is independant to any HandleScope, valid until cleared

```
    static void AddRule(const v8::FunctionCallbackInfo<v8::Value>&);
```

```
    static void ScanFile(const v8::FunctionCallbackInfo<v8::Value>&);
public:
    static void Init(v8::Handle<v8::Object>);
    static int yrstatus;
};
```

The Persistent storage will serve us for the YaraNode initialization from within the Nodejs entry point:

```
#include "yaranode.h"


using namespace v8;


void addrulecb(int error, const char *, int line,
        const char *message, void *pprivate) {
    Isolate *isolate = Isolate::GetCurrent();
    if (message)

       isolate->ThrowException(Exception::TypeError(String::NewFromUt
f8(
           isolate, message)));
}
Persistent<Function> YaraNode::constructor;


YaraNode::YaraNode() {
    yrstatus = yr_initialize();
```

NodeJS

```
    if (yrstatus == ERROR_SUCCESS) {
 yr_compiler_create(&yc);
        yr_compiler_set_callback(yc, addrulecb, NULL);
    }
}


YaraNode::~YaraNode() {
    if (yrstatus == ERROR_SUCCESS) {
        yr_compiler_destroy(yc);
        yr_finalize();
    }
}


void YaraNode::New(const FunctionCallbackInfo<Value> &args) {
    Isolate *isolate;
    Local<Function> ctor;
    isolate = Isolate::GetCurrent();
    HandleScope scope(isolate);
```

=> A HandleScope is responsible for all following local handles allocations

```
    if (args.IsConstructCall()) {
```

=> var yr = new YaraNode();

```
        YaraNode *ynode = new YaraNode();
        if (ynode->yrstatus != ERROR_SUCCESS)
```

```
            isolate->ThrowException(Exception::TypeError(
                String::NewFromUtf8(isolate, "yara could not be in-
stantiated")));
  ynode->Wrap(args.This());
```
=> Here we wrap our YaraNode and can be unwrap as will as we ll see slighty later

```
        args.GetReturnValue().Set(args.This());
```
=> We return basically the wrapped yaranode object to the javascript caller

```
    } else {
```
=> YaraNode called as classic function

```
        ctor = Local<Function>::New(isolate, constructor);
```
=> We use here our persistent storage to instantiate our YaraNode instance

```
        args.GetReturnValue().Set(ctor->NewInstance());
    }
}


void YaraNode::AddRule(const FunctionCallbackInfo<Value> &args) {
    Isolate *isolate;
    int yrc = 0;


    isolate = Isolate::GetCurrent();
    HandleScope scope(isolate);
    YaraNode *ynode = ObjectWrap::Unwrap<YaraNode>(args.Holder());
```
=> Here we unwrap to access a YaraNode object field

```
 if (args.Length() > 0) {
        int i, r;
```

NodeJS

```
  for (i = 0; i < args.Length(); i ++) {
```
=> addRule method, from nodejs script, is called like this addRule(<rule1>,...,<ruleN>);

```
            if (args[i]->IsString()) {
                const char *rule;
                String::Utf8Value rrstr(args[i]->ToString());
                rule = *rrstr;
                r = yr_compiler_add_string(ynode->yc, rule, 0);
                if (r == 0)
                    ynode->yrrules ++;
                yrc += r;
            }
        }
    }

    args.GetReturnValue().Set(Number::New(isolate, yrc));
}

void YaraNode::ScanFile(const FunctionCallbackInfo<Value>& args) {
    Isolate *isolate;
    int yrscan = 0;

    isolate = Isolate::GetCurrent();
    HandleScope scope(isolate);
```

NodeJS

```
    YaraNode *ynode = ObjectWrap::Unwrap<YaraNode>(args.Holder());
    if (args.Length() == 1 && args[0]->IsString()) {
        YR_RULES *rules = 0;
        const char *filepath;
        if (ynode->yrrules > 0 &&
            yr_compiler_get_rules(ynode->yc, &rules) == ERROR_SUC-
CESS) {
            String::Utf8Value fstr(args[0]->ToString());
            filepath = *fstr;
            yrscan = yr_rules_scan_file(rules, filepath, 0,
                    NULL, NULL, 10);
        }
    }
 args.GetReturnValue().Set(Number::New(isolate, yrscan));
}
void YaraNode::Init(Handle<Object> exports) {
    Local<FunctionTemplate> temp;
    Isolate *isolate;

    isolate = Isolate::GetCurrent();
    temp = FunctionTemplate::New(isolate, New);
    temp->SetClassName(String::NewFromUtf8(isolate, "YaraNode")); =>
```

From within a nodejs script, the class will have this name, we could have named it differently if necessary

```
  NODE_SET_PROTOTYPE_METHOD(temp, "addRule", YaraNode::AddRule);
```

=> As the single functions with NODE_SET_METHOD, we expose our methods via this macro

```
    NODE_SET_PROTOTYPE_METHOD(temp, "scanFile", YaraNode::ScanFile);


    constructor.Reset(isolate, temp->GetFunction());
```

=> We clear the Persistent storage for each YaraNode instantation

```
    exports->Set(String::NewFromUtf8(isolate, "YaraNode"),
            temp->GetFunction());
}


void YaraInit(Handle<Object> exports) {
    YaraNode::Init(exports);
}


NODE_MODULE(yara, YaraInit)
    temp->InstanceTemplate()->SetInternalFieldCount(2);
```

We could test this module via this simple nodejs script.

```
var sm = require('./build/Release/yaranode');
var yr = new sm.YaraNode();


try {
    var c = yr.addRule("<rule 1>",...);
    ...
```

```
var s = yr.scanFile("<file path>");

    ...

} catch (ex) {

    console.log(ex);

}
```

This is a simple example and can of course be greatly improved but that might give you some ideas about the possibilities. On several known repositories, a significant amount of native nodejs projects already exist that use some popular components (like node geoip, for example). I hope this article is able to motivate you enough to start building your own nodejs modules.

Cloud service from a developer point of view

In this article, we will have an overview of writing a cloud service. Various ways exist to achieve your goals but we will focus on one that is memory efficient, multiplatform (POSIX systems), multi language (from C++ to Erlang), and reasonably fast. It is Apache Thrift. I recently fully wrote a cloud service and it worked reliably.

To illustrate this, we will make a basic remote file handler, the server is written in C++ and the client written in Python as an example.

1. Describing the service

Our server will be able to deliver three different services, listing files or directories, deleting or moving a file. Thrift is an IDL (Interface Definition Language) based framework, hence you describe your service via an abstract generic language and the Thrift compiler will generate the necessary code per programming language. The basic Thrift types are all we find in common in all languages, byte, binary, integer (i16/32/64), double, boolean, string, some containers as hashmap, sets or lists. For those familiar with C and or C++ we can define an atomic file with a “struct”:

```
struct file {

        1: string name

}
```

The number means the index of the name's field. A file in a UNIX system can have several types, not necessarily a regular file but a device, a socket and so forth.

So, let's enumerate each type we might need to identify the files, again “a la” C/C++:

```
enum file_type {
        FILE = 0,
        DEVICE = 1,
        SOCKET = 2,
        SYMLINK = 3,
        DIRECTORY = 4
}
...
struct file {
        1: file_type type
        2: string name
}
```

What if we store some file attributes like the size, the permissions bits ... ? Thrift allows you to set a struct inside a struct without problems as you can see below:

```
struct file_attribute {
        1: i32 uid
        2: i32 gid
        3: i16 mask
        4: i64 size
        5: string strmask
}
```

```
struct file {
        1: file_type type
        2: file_attribute attr
        3: string name
}
```

Now, we can start to describe the three Thrift “services” as below; for the first, we would like to return a map of files and for the sake of shortening, we “typedef” it as below:

```
typedef map<string, file> file_list
```

In addition, for our services we would like to throw an exception in case something went wrong. A Thrift exception is very similar to a struct:

```
exception file_exception {
        1: i16 code
        2: string msg
}
```

If we do not write the required keyword, a field is then optional. If you re not sure for future development that a field ought to be required, I'd suggest to leave it optional as the clients would stop working if the previously required field was suddenly optional in the server's side.

```
service file_service {
        file_list eforensics_ls(1: required string path) throws (1: file_exception ex),
        i16 eforensics_rm(1: required string path) throws (1: file-exception ex),
        i16 eforensics_mv(1: required string src, 2: required string dst) throws (1: file_exception ex),
}
```

Cloud

Above all of that we might need to customize the language namespace to organize and avoid conflicts; for Java and C++ developers, for example, it is pretty well known. The namespace will be translated as well in the target language's logic:

```
namespace cpp eforensics.cloud

namespace py eforensics.cloud
```

The first will produce the usual C++'s namespace as:

```
namespace eforensics { namespace cloud {
```

Whereas the latter will make the `eforensics/cloud` Python module.

In the end, the thrift file might look like this:

```
namespace cpp eforensics.cloud

namespace py eforensics.cloud


enum file_type {

    FILE = 0,

    DEVICE = 1,

    SOCKET = 2,

    SYMLINK = 3,

    DIRECTORY = 4

}


struct file_attribute {
```

```
  1: i32 uid

    2: i32 gid

    3: i16 mask

    4: i64 size

    5: string strmask

}


struct file {

    1: file_type type

    2: file_attribute attr

    3: string name

}


typedef map<string, file> file_list


exception file_exception {

    1: i16 code

    2: string msg

}


service file_service {

    file_list eforensics_ls(1: required string path) throws (1:
file_exception ex),
```

```
     i16 eforensics_rm(1: required string path) throws (1: file_excep-
tion ex),

    i16 eforensics_mv(1: required string src, 2: required string dst)
throws (1: file_exception ex),

}
```

2. Generating the code

Once the service is defined, we can now use the thrift compiler like this:

```
$ thrift -gen cpp eforensics.thrift

$ ls

eforensics.thrifts gen-cpp

$ thrift -gen py eforensics.thrift

...
```

In the C++ version, we realize that a skeleton server was generated as well. We will use it to implement our service!

```
// This autogenerated skeleton file illustrates how to build a server.

// You should copy it to another filename to avoid overwriting it.


#include "file_service.h"

#include <thrift/protocol/TBinaryProtocol.h>

#include <thrift/server/TSimpleServer.h>

#include <thrift/transport/TServerSocket.h>

#include <thrift/transport/TBufferTransports.h>
```

Cloud

```
using namespace ::apache::thrift;

using namespace ::apache::thrift::protocol;

using namespace ::apache::thrift::transport;

using namespace ::apache::thrift::server;


using boost::shared_ptr;


using namespace  ::eforensics::cloud;


class file_serviceHandler : virtual public file_serviceIf {

 public:

  file_serviceHandler() {

    // Your initialization goes here

  }


  void eforensics_ls(file_list& _return, const std::string& path) {

  // Your implementation goes here

    printf("eforensics_ls\n");

  }

  int16_t eforensics_rm(const std::string& path) {

   // Your implementation goes here

    printf("eforensics_rm\n");

  }
```

```
 int16_t eforensics_mv(const std::string& src, const std::string&
dst) {
   // Your implementation goes here
     printf("eforensics_mv\n");
   }


};


int main(int argc, char **argv) {
  int port = 9090;
  shared_ptr<file_serviceHandler> handler(new file_serviceHandler());
  shared_ptr<TProcessor> processor(new file_serviceProcessor(han-
dler));
  shared_ptr<TServerTransport> serverTransport(new TServerSock-
et(port));
  shared_ptr<TTransportFactory> transportFactory(new TBufferedTrans-
portFactory());
  shared_ptr<TProtocolFactory> protocolFactory(new TBinaryProtocolFac-
tory());


  TSimpleServer server(processor, serverTransport, transportFactory,
protocolFactory);
  server.serve();
  return 0;
}
```

Now it is up to us to implement the three services. Let's start with the simplest, removing a file with the famous C function unlink.

```
nt16_t eforensics_rm(const std::string& path) {
    if (unlink(path.c_str()) == -1) {
        return -1;
    }

    return 0;
}
```

To improve it, we could make sure the file is a regular file, otherwise return the exception we set earlier in the thrift IDL file.

```
void create_stat(struct stat &s, const string &path) {
      if (path.size() > MAXPATHLEN) {
          string msg = "Name too long ";
          msg += path;
          f.__set_code(-1);
          f.__set_msg(msg);
          throw f;
      }
 ::memset(&s, 0, sizeof(s));

      if (stat(path.c_str(), &s) == -1) {
          string msg = "Could not stat ";
```

```
            msg += path;
            f.__set_code(-1);
            f.__set_msg(msg);
            throw f;
        }
    }
....
struct stat s;
    create_stat(s, path);
    mode_t m = s.st_mode;

    if ((m & S_IFMT) != S_IFREG) {
        string msg = "Only files can be removed";
        f.__set_code(-1);
        f.__set_msg(msg);
        throw f;
    }

    if (unlink(path.c_str()) == -1) {
        string msg = "Could not remove ";
        msg += path;
        msg += ": ";
        msg += strerror(errno);
```

```
        f.__set_code(-1);
        f.__set_msg(msg);
        throw f;
    }
...
```

In a similar manner, we can do the move function.

```
int16_t eforensics_mv(const std::string& src, const std::string& dst)
{
    struct stat s;
    create_stat(s, src);
    mode_t m = s.st_mode;

    if ((m & S_IFMT) != S_IFREG) {
        string msg = "Only files can be moved";
        f.__set_code(-1);
        f.__set_msg(msg);
        throw f;
    }
  if (rename(src.c_str(), dst.c_str()) == -1) {
        string msg = "Could not move ";
        msg += src;
        msg += " to ";
```

```
        msg += dst;
        msg += ": ";
    msg += strerror(errno);
        f.__set_code(-1);
        f.__set_msg(msg);
        throw f;
    }


    return 0;
}
```

Then the last service, listing files or directories. Previously, we defined several types of files and their attributes, hence we'll once again rely on the stat function:

```
int16_t ls_add_entry(file_list & _return, const string path) {
        bool isDir = false;
        struct stat s;
        create_stat(s, path);
        mode_t m = s.st_mode;


        file fl;
        fl.attr.uid = s.st_uid;
        fl.attr.gid = s.st_gid;
        fl.attr.size = s.st_size;
```

```
        fl.name = path;
    if ((m & S_IFMT) == S_IFBLK || (m & S_IFMT) == S_IFCHR ||
                    (m & S_IFMT) == S_IFIFO) {
            fl.type = file_type::type::DEVICE;
        } else if ((m & S_IFMT) == S_IFSOCK) {
            fl.type = file_type::type::SOCKET;
        } else if ((m & S_IFMT) == S_IFLNK) {
            fl.type = file_type::type::SYMLINK;
        } else if ((m & S_IFMT) == S_IFREG) {
            fl.type = file_type::type::FILE;
        } else if ((m & S_IFMT) == S_IFDIR && (((m & S_IFMT) & S_IFLNK)
!= S_IFLNK)) {
            fl.type = file_type::type::DIRECTORY;
            isDir = true;
        }

        fl.attr.mask = 0;
        if (m & S_IWUSR)
            fl.attr.mask |= 0x400;
        if (m & S_IRUSR)
            fl.attr.mask |= 0x200;
        if (m & S_IXUSR)
            fl.attr.mask |= 0x100;
```

```
if (m & S_IWGRP)
        fl.attr.mask |= 0x040;
    if (m & S_IRGRP)
        fl.attr.mask |= 0x020;
    if (m & S_IXGRP)
        fl.attr.mask |= 0x010;
    if (m & S_IWOTH)
        fl.attr.mask |= 0x004;
    if (m & S_IROTH)
        fl.attr.mask |= 0x002;
    if (m & S_IXOTH)
        fl.attr.mask |= 0x001;


char strmask[5];
sprintf(strmask, "%04x", fl.attr.mask);
fl.attr.strmask = string(strmask);


_return[path] = fl;
if (isDir) {
    DIR *dir = opendir(path.c_str());
    if (dir == NULL) {
        return -1;
    }
```

```
struct dirent entry, *result = NULL;

        // We could have just used readdir but we might need to run it

        // in multi thread context ..

        while (readdir_r(dir, &entry, &result) == 0) {

            if (result == NULL)

                break;

            if (strcmp(".", result->d_name) == 0 ||

                strcmp("..", result->d_name) == 0)

                continue;

            string rpath = path;

            if (rpath[path.size() -1] != '/')

                rpath += "/";

            rpath += result->d_name;

            ls_add_entry(_return, rpath);

        }


        closedir(dir);

    }


    return 0;

  }

...
```

```
// It is better in terms of an interface, in the case of C++, to not return
// a map as the IDL defined
void eforensics_ls(file_list& _return, const std::string& path) {
      _return.clear();

      ls_add_entry(_return, path);
}
...
```

We are nearly done, let's compile the code.

```
$ c++ -std=c++11 -g -O2 -I. -I/usr/local/include -o
eforensics_constants.o -c eforensics_constants.cpp

$ c++ -std=c++11 -g -O2 -I. -I/usr/local/include -o
eforensics_types.o -c eforensics_types.cpp

$ c++ -std=c++11 -g -O2 -I. -I/usr/local/include -o file_service.o -c
file_service.cpp

$ c++ -Wall -std=c++11 -g -O2 -I. -I/usr/local/include -o
file_service_server.o -c file_service_server.skeleton.cpp

$ c++ -std=c++11 -g -O2 -o eforensics_file_service
eforensics_constants.o eforensics_types.o file_service.o
file_service_server.o -Wl,-rpath,/usr/local/lib -L/usr/local/lib
-lthrift
```

If you execute the final executable, it will listen via the 9090 port and if you generated Python's version, for example, it should have generated a sample client:

```
$ ./file_service-remote -h 192.168.1.11:9090 eforensics_ls /tmp
```

```
{ '/tmp': file(type=4, attr=file_attribute(gid=0, mask=None, uid=0,
strmask='0777', size=None), name='/tmp'),

  '/tmp/.ICE-unix': file(type=4, attr=file_attribute(gid=0,
mask=None, uid=0, strmask='0777', size=None), name='/tmp/.ICE-unix'),

  '/tmp/.ICE-unix/1997': file(type=2, attr=file_attribute(gid=1000,
mask=None, uid=1000, strmask='0777', size=None),
name='/tmp/.ICE-unix/1997'),

  '/tmp/.X0-lock': file(type=0, attr=file_attribute(gid=0, mask=None,
uid=0, strmask='0222', size=None), name='/tmp/.X0-lock'),

  '/tmp/.X11-unix': file(type=4, attr=file_attribute(gid=0,
mask=None, uid=0, strmask='0777', size=None), name='/tmp/.X11-unix'),

  '/tmp/.X11-unix/X0': file(type=2, attr=file_attribute(gid=0,
mask=None, uid=0, strmask='0777', size=None),
name='/tmp/.X11-unix/X0'),

  '/tmp/.vbox-dcarlier-ipc': file(type=4, attr=file_attri-
bute(gid=1000, mask=None, uid=1000, strmask='0700', size=None),
name='/tmp/.vbox-dcarlier-ipc'),

  '/tmp/.vbox-dcarlier-ipc/ipcd': file(type=2, attr=file_attri-
bute(gid=1000, mask=None, uid=1000, strmask='0700', size=None),
name='/tmp/.vbox-dcarlier-ipc/ipcd'),

  '/tmp/.vbox-dcarlier-ipc/lock': file(type=0, attr=file_attri-
bute(gid=1000, mask=None, uid=1000, strmask='0600', size=None),
name='/tmp/.vbox-dcarlier-ipc/lock'),

  '/tmp/config-err-tu3hNl': file(type=0, attr=file_attri-
bute(gid=1000, mask=None, uid=1000, strmask='0600', size=None),
name='/tmp/config-err-tu3hNl'),

  '/tmp/unity_support_test.0': file(type=0, attr=file_attri-
bute(gid=1000, mask=None, uid=1000, strmask='0662', size=None),
name='/tmp/unity_support_test.0')}
```

Cloud

We can have a quick look at how the Python version is made:

```
...
if http:
  transport = THttpClient.THttpClient(host, port, uri)
else:
  socket = TSSLSocket.TSSLSocket(host, port, validate=False) if ssl
else TSocket.TSocket(host, port)
  if framed:
    # In this mode, the message is fully read no flush is required
    transport = TTransport.TFramedTransport(socket)
  else:
    transport = TTransport.TBufferedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(transport)
client = file_service.Client(protocol)
transport.open()
...
# Pretty straightforward to call each server method as you can see
if cmd == 'eforensics_ls':
  if len(args) != 1:
    print('eforensics_ls requires 1 args')
    sys.exit(1)
  pp.pprint(client.eforensics_ls(args[0],))


...
```

```
elif cmd == 'eforensics_rm':
  if len(args) != 1:
    print('eforensics_rm requires 1 args')
    sys.exit(1)
  pp.pprint(client.eforensics_rm(args[0],))


elif cmd == 'eforensics_mv':
  if len(args) != 2:
    print('eforensics_mv requires 2 args')
    sys.exit(1)
  pp.pprint(client.eforensics_mv(args[0],args[1],))


else:
  print('Unrecognized method %s' % cmd)
  sys.exit(1)


transport.close()
```

If we come back to the C++ server's code, the skeleton's generated code uses a TsimpleServer, which is perfect to start with but is monothread. I'd suggest the TThreadPoolServer (more efficient than the TThreadedServer) or the TNonBlockingServer instead and to at least add a signal handler to terminate the server properly. The TthreadPoolServer's version might look like this:

```
...
  signal(SIGINT, servsighandler);
```

```
signal(SIGQUIT, servsighandler);

    signal(SIGPIPE, servsighandler);



    try {

        shared_ptr<TProcessor> processor(new cloud_service_adminProc-
essor(handler));

        shared_ptr<TServerTransport> serverTransport(new TServerSock-
et(port));

        shared_ptr<TTransportFactory> transportFactory(new TBuffered-
TransportFactory);

        shared_ptr<TProtocolFactory> protocolFactory(new TBinaryProto-
colFactory());



        threadManager =
ThreadManager::newSimpleThreadManager(workers);

        shared_ptr<PosixThreadFactory> threadFactory(new PosixThread-
Factory());

        threadManager->threadFactory(threadFactory);

        threadManager->start();



        std::clog << "server is starting" << std::endl;

        nserver = shared_ptr<TServer>(new TThreadPoolServer(proces-
sor, serverTransport, transportFactory, protocolFactory, threadMan-
ager));

        nserver->serve();

    } catch (std::exception &e) {
```

```
        std::clog << "An error occured: " << e.what() << std::endl;
    }


...
```

3. Conclusion

Apache Thrift works well indeed in most POSIX systems, I've made the full example server part of a Linux machine and tested with FreeBSD and Linux. The client was called on a remote FreeBSD machine.

An alternative version exists remade by Facebook called fbthrift which works fully only on Linux but the code generated is superior and this version in general has proved to be more efficient in term of memory usage at least. There is also Google Protocol Buffer that performs better than the two above but has fewer languages supported (officially). You have to write the client / server code on your own, though. So based on your own criteria and restrictions, one of these might fit better for your own case.

About the Author

Born in France, David Carlier mainly lived around or in Paris. Like many children of his generation, he got hooked on computers before 10 years old during the eighties, started with Thomson MO5 then later on, an Amstrad CPC 6128 with long hours of BASIC programming during the weekends and video game playing.

Several years later, he bought a PC with his first earnings to do both development and some electronic music as a hobby. He started his career early in 2001 for the CNAM, which was where he earned his Bachelor degree, studying Ada and distributed systems. He started professionally working with Microsoft Windows NT and 95 until one day at work, he found a CD-ROM of a Linux distribution (Mandrake, the ancestor of the now defunct Mandriva), installed it on his desktop, replacing Windows 95 and was immediately attracted by several factors.

Indeed, all servers around moved on to various Unixes, from SunOS to Linux RedHat, thus having similar commands, software (web and mail servers mostly) and programming languages (Perl, PHP, Java and C mainly) made everthing easier for development. What amazed him more is the fact that all the source code was provided, representing long hours of efforts from awesome engineers ...

This first step into the open source led to increase the curiosity to try other distributions, from Debian to various flavors of RedHat, once Fedora and CentOS started their existence, to finally end up with Slackware, which he found more effective, stricter to the original “UNIX path”. Furthermore, other alternatives came into play, like QNX when the free version on x86 still existed, but more importantly, the BSD flavors, starting with FreeBSD, which he installed in most of the servers at work. Rapidly attracted by the more strict UNIX approach, the real separation between the base and third party software, as this time using only its famous port system and its unique ncurses options menu choices; but the experience gained with Slackware helped quite a lot to take a grip on, as at this time few software options were available, hence the tryptic “./configure/make/make install” and dependencies management were well known already.

About the Author

OpenBSD and NetBSD really came along later, although the attraction for those really came several years later. Becoming a parent, he stopped for couple of years the usage and study of open source components during his spare time, and moved to different companies, as well (ISP, financial transactions, web platforms ...), as an employee or for a couple of years as contractor, but with either Windows or Linux, trying to use open source software whenever possible.

During the last quarter of 2012, he got an opportunity to work in the heart of the Republic of Ireland, Dublin, and the daughter had grown up so he was able to get back more spare time, plus the fact that in this position, FreeBSD is widely used; that was more than enough to get interested in it again. Indeed, contributing to open source projects provided a certain amount of benefits, contributing to useful components widely used, being a part of a larger purpose than alone, getting to know a wide range of persons, stretching our own knowledge while strengthening up ourselves. Hence, he started to contribute modestly to ISC Bind 10 (now split between Kea and Bundy projects), which made him understand how open source teams would possibly work, then later on contribute both professionally and personally to Haproxy, various flavors of FreeBSD (mainly userland and third party software fixes), for a short while to DragonflyBSD, until he hooked up with OpenBSD, attracted by the focus on the quality code and security features available on base, where he focuses on improving the multimedia experience in general (video games, music production software, ...), which highlights more the fact that OpenBSD has the capacity to be used as a day-to-day operating system, maintaining a couple of packages, while trying to push some existing port patches to mainstream projects. At last, contributing time to time to OpenBSD's code base. Apart from OpenBSD related projects, some modest contributions are done to various projects, like the official PHP interpreter or the promising web server h2o.